//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace IR

{

class LabelInstr;

}

enum JsNativeValueType;

namespace Js

{

struct ResumeYieldData;

#define DeclareExceptionPointer(ep) \

EXCEPTION\_RECORD ep##er; \

CONTEXT ep##c; \

EXCEPTION\_POINTERS ep = {&ep##er, &ep##c};

#define TYPEOF\_ERROR\_HANDLER\_CATCH(scriptContext, var) \

} \

catch (Js::JavascriptExceptionObject \*exceptionObject) \

{ \

Js::Var errorObject = exceptionObject->GetThrownObject(nullptr); \

if (errorObject != nullptr && Js::JavascriptError::Is(errorObject)) \

{ \

HRESULT hr = Js::JavascriptError::GetRuntimeError(Js::RecyclableObject::FromVar(errorObject), nullptr); \

if (JavascriptError::GetErrorNumberFromResourceID(JSERR\_Property\_CannotGet\_NullOrUndefined) == (long)hr \

|| JavascriptError::GetErrorNumberFromResourceID(JSERR\_UseBeforeDeclaration) == (long)hr) \

{ \

if (scriptContext->IsInDebugMode()) \

{ \

JavascriptExceptionOperators::ThrowExceptionObject(exceptionObject, scriptContext, true); \

} \

else \

{ \

throw exceptionObject; \

} \

} \

} \

var = scriptContext->GetLibrary()->GetUndefined();

#define TYPEOF\_ERROR\_HANDLER\_THROW(scriptContext, var) \

} \

if (scriptContext->IsUndeclBlockVar(var)) \

{ \

Assert(scriptContext->GetConfig()->IsLetAndConstEnabled()); \

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UseBeforeDeclaration); \

}

#define BEGIN\_TYPEOF\_ERROR\_HANDLER(scriptContext) \

try { \

Js::JavascriptExceptionOperators::AutoCatchHandlerExists autoCatchHandlerExists(scriptContext); \

class AutoCleanup \

{ \

private: \

ScriptContext \*const scriptContext; \

public: \

AutoCleanup(ScriptContext \*const scriptContext) : scriptContext(scriptContext) \

{ \

if (scriptContext->IsInDebugMode()) \

{ \

scriptContext->GetDebugContext()->GetProbeContainer()->SetThrowIsInternal(true); \

} \

} \

~AutoCleanup() \

{ \

if (scriptContext->IsInDebugMode()) \

{ \

scriptContext->GetDebugContext()->GetProbeContainer()->SetThrowIsInternal(false); \

} \

} \

} autoCleanup(scriptContext);

#define END\_TYPEOF\_ERROR\_HANDLER(scriptContext, var) \

TYPEOF\_ERROR\_HANDLER\_CATCH(scriptContext, var) \

TYPEOF\_ERROR\_HANDLER\_THROW(scriptContext, var)

#define BEGIN\_PROFILED\_TYPEOF\_ERROR\_HANDLER(scriptContext) \

BEGIN\_TYPEOF\_ERROR\_HANDLER(scriptContext)

#define END\_PROFILED\_TYPEOF\_ERROR\_HANDLER(scriptContext, var, functionBody, inlineCacheIndex) \

TYPEOF\_ERROR\_HANDLER\_CATCH(scriptContext, var) \

functionBody->GetDynamicProfileInfo()->RecordFieldAccess(functionBody, inlineCacheIndex, var, FldInfo\_NoInfo); \

TYPEOF\_ERROR\_HANDLER\_THROW(scriptContext, var)

class JavascriptOperators /\* All static \*/

{

// Methods

public:

static void FreeTemp(Var aValue);

static BOOL IsArray(Var instanceVar);

static BOOL IsConstructor(Var instanceVar);

static BOOL IsConcatSpreadable(Var instanceVar);

static Var ToObject(Var aRight,ScriptContext\* scriptContext);

static Var ToWithObject(Var aRight, ScriptContext\* scriptContext);

static Var OP\_LdCustomSpreadIteratorList(Var aRight, ScriptContext\* scriptContext);

static Var ToNumber(Var aRight,ScriptContext\* scriptContext);

static Var ToNumberInPlace(Var aRight,ScriptContext\* scriptContext, JavascriptNumber\* result);

#ifdef \_M\_IX86

static Var Int32ToVar(int32 value, ScriptContext\* scriptContext);

static Var Int32ToVarInPlace(int32 value, ScriptContext\* scriptContext, JavascriptNumber \*result);

static Var UInt32ToVar(uint32 value, ScriptContext\* scriptContext);

static Var UInt32ToVarInPlace(uint32 value, ScriptContext\* scriptContext, JavascriptNumber \*result);

#endif

static Var OP\_FinishOddDivBy2(uint32 value, ScriptContext \*scriptContext);

static Var OP\_ApplyArgs(Var func,Var instance,\_\_in\_xcount(8)void\*\* stackPtr,CallInfo callInfo,ScriptContext\* scriptContext);

static Var Typeof(Var var, ScriptContext\* scriptContext);

static Var TypeofFld(Var instance, PropertyId propertyId, ScriptContext\* scriptContext);

static Var TypeofRootFld(Var instance, PropertyId propertyId, ScriptContext\* scriptContext);

static Var TypeofElem(Var instance, Var index, ScriptContext\* scriptContext);

static Var TypeofElem\_UInt32(Var instance, uint32 index, ScriptContext\* scriptContext);

static Var TypeofElem\_Int32(Var instance, int32 index, ScriptContext\* scriptContext);

static Var Delete(Var var, ScriptContext\* scriptContext);

static JavascriptString \* Concat3(Var aLeft, Var aCenter, Var aRight, ScriptContext \* scriptContext);

static JavascriptString \* NewConcatStrMulti(Var a1, Var a2, uint count, ScriptContext \* scriptContext);

static void SetConcatStrMultiItem(Var concatStr, Var str, uint index, ScriptContext \* scriptContext);

static void SetConcatStrMultiItem2(Var concatStr, Var str1, Var str2, uint index, ScriptContext \* scriptContext);

static BOOL Equal(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL Equal\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL Greater(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL Greater\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL GreaterEqual(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL GreaterEqual\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL Less(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL Less\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL LessEqual(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL LessEqual\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL NotEqual(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL NotEqual\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL StrictEqual(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL StrictEqualString(Var aLeft, Var aRight);

static BOOL StrictEqualEmptyString(Var aLeft);

static BOOL NotStrictEqual(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL HasOwnProperty(Var instance, PropertyId propertyId, ScriptContext \* requestContext);

static BOOL GetOwnProperty(Var instance, PropertyId propertyId, Var\* value, ScriptContext\* requestContext);

static BOOL GetOwnAccessors(Var instance, PropertyId propertyId, Var\* getter, Var\* setter, ScriptContext \* requestContext);

static BOOL EnsureProperty(Var instance, PropertyId propertyId);

static void OP\_EnsureNoRootProperty(Var instance, PropertyId propertyId);

static void OP\_EnsureNoRootRedeclProperty(Var instance, PropertyId propertyId);

static void OP\_ScopedEnsureNoRedeclProperty(FrameDisplay \*pDisplay, PropertyId propertyId, Var instanceDefault);

static Var GetOwnPropertyNames(Var instance, ScriptContext \*scriptContext);

static Var GetOwnPropertySymbols(Var instance, ScriptContext \*scriptContext);

static Var GetOwnPropertyKeys(Var instance, ScriptContext \*scriptContext);

static Var GetOwnEnumerablePropertyNames(Var instance, ScriptContext \*scriptContext);

static Var GetOwnEnumerablePropertyNamesSymbols(Var instance, ScriptContext \*scriptContext);

static BOOL GetOwnPropertyDescriptor(RecyclableObject\* obj, PropertyId propertyId, ScriptContext\* scriptContext, PropertyDescriptor\* propertyDescriptor);

static BOOL GetOwnPropertyDescriptor(RecyclableObject\* obj, JavascriptString\* propertyKey, ScriptContext\* scriptContext, PropertyDescriptor\* propertyDescriptor);

static BOOL IsPropertyUnscopable (Var instanceVar, PropertyId propertyId);

static BOOL IsPropertyUnscopable (Var instanceVar, JavascriptString \*propertyString);

template<bool unscopables>

static BOOL HasProperty\_Impl(RecyclableObject\* instance, PropertyId propertyId);

static BOOL HasPropertyUnscopables(RecyclableObject\* instance, PropertyId propertyId);

static BOOL HasProperty(RecyclableObject\* instance, PropertyId propertyId);

static BOOL HasRootProperty(RecyclableObject\* instance, PropertyId propertyId);

static BOOL HasProxyOrPrototypeInlineCacheProperty(RecyclableObject\* instance, PropertyId propertyId);

static BOOL HasProxyInPrototypeChain(RecyclableObject\* instance);

template<typename PropertyKeyType>

static BOOL GetPropertyWPCache(Var instance, RecyclableObject\* propertyObject, PropertyKeyType propertyKey, Var\* value, ScriptContext\* requestContext, PropertyString \* propertyString);

static BOOL GetPropertyUnscopable(Var instance, RecyclableObject\* propertyObject, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info=NULL);

static Var GetProperty(RecyclableObject\* instance, PropertyId propertyId, ScriptContext\* requestContext, PropertyValueInfo\* info = NULL);

static BOOL GetProperty(RecyclableObject\* instance, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info = NULL);

static Var GetProperty(Var instance, RecyclableObject\* propertyObject, PropertyId propertyId, ScriptContext\* requestContext, PropertyValueInfo\* info = NULL);

static BOOL GetProperty(Var instance, RecyclableObject\* propertyObject, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info = NULL);

static BOOL GetPropertyObject(Var instance, ScriptContext \* scriptContext, RecyclableObject\*\* propertyObject);

static BOOL GetRootProperty(Var instance, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info = NULL);

static Var GetRootProperty(RecyclableObject\* instance, PropertyId propertyId, ScriptContext\* requestContext, PropertyValueInfo\* info = NULL);

static Var GetPropertyReference(RecyclableObject\* instance, PropertyId propertyId, ScriptContext\* requestContext);

static BOOL GetPropertyReference(RecyclableObject\* instance, PropertyId propertyId, Var\* value,ScriptContext\* requestContext, PropertyValueInfo\* info = NULL);

static BOOL GetPropertyReference(Var instance, RecyclableObject\* propertyObject, PropertyId propertyId, Var\* value,ScriptContext\* requestContext, PropertyValueInfo\* info = NULL);

static BOOL GetRootPropertyReference(RecyclableObject\* instance, PropertyId propertyId, Var\* value,ScriptContext\* requestContext, PropertyValueInfo\* info = NULL);

template<typename PropertyKeyType>

static BOOL SetPropertyWPCache(Var instance, RecyclableObject\* object, PropertyKeyType propertyKey, Var newValue, ScriptContext\* requestContext, PropertyString \* propertyString, PropertyOperationFlags flags);

static BOOL SetPropertyUnscopable(Var instance, RecyclableObject\* receiver, PropertyId propertyId, Var newValue, PropertyValueInfo \* info, ScriptContext\* requestContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL SetProperty(Var instance, RecyclableObject\* object, PropertyId propertyId, Var newValue, ScriptContext\* requestContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL SetProperty(Var instance, RecyclableObject\* receiver, PropertyId propertyId, Var newValue, PropertyValueInfo \* info, ScriptContext\* requestContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL SetRootProperty(RecyclableObject\* instance, PropertyId propertyId, Var newValue, PropertyValueInfo \* info, ScriptContext\* requestContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL GetAccessors(RecyclableObject\* instance, PropertyId propertyId, ScriptContext\* requestContext, Var\* getter, Var\* setter);

static BOOL SetAccessors(RecyclableObject\* instance, PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL InitProperty(RecyclableObject\* instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL DeleteProperty(RecyclableObject\* instance, PropertyId propertyId, PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

static BOOL DeletePropertyUnscopables(RecyclableObject\* instance, PropertyId propertyId, PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

template<bool unscopables>

static BOOL DeleteProperty\_Impl(RecyclableObject\* instance, PropertyId propertyId, PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

static TypeId GetTypeId(Var instance);

static BOOL IsObject(Var instance);

static BOOL IsExposedType(TypeId typeId);

static BOOL IsObjectType(TypeId typeId);

static BOOL IsObjectOrNull(Var instance);

static BOOL IsUndefinedOrNullType(TypeId);

static BOOL IsSpecialObjectType(TypeId typeId);

static BOOL IsJsNativeObject(Var instance);

static BOOL IsUndefinedObject(Var instance);

static BOOL IsUndefinedObject(Var instance, ScriptContext \*scriptContext);

static BOOL IsUndefinedObject(Var instance, RecyclableObject \*libraryUndefined);

static BOOL IsUndefinedObject(Var isntance, JavascriptLibrary\* library);

static BOOL IsAnyNumberValue(Var instance);

static BOOL IsIterable(RecyclableObject\* instance, ScriptContext\* scriptContext);

static BOOL IsClassConstructor(Var instance);

static BOOL HasOwnItem(RecyclableObject\* instance, uint32 index);

static BOOL HasItem(RecyclableObject\* instance, uint32 index);

static BOOL HasItem(RecyclableObject\* instance, uint64 index);

static BOOL GetOwnItem(RecyclableObject\* instance, uint32 index, Var\* value, ScriptContext\* requestContext);

static BOOL GetItem(RecyclableObject\* instance, uint64 index, Var\* value, ScriptContext\* requestContext);

static BOOL GetItem(RecyclableObject\* instance, uint32 index, Var\* value, ScriptContext\* requestContext);

static BOOL GetItem(Var instance, RecyclableObject\* propertyObject, uint32 index, Var\* value, ScriptContext\* requestContext);

static BOOL GetItemReference(RecyclableObject\* instance, uint32 index, Var\* value, ScriptContext\* requestContext);

static BOOL GetItemReference(Var instance, RecyclableObject\* propertyObject, uint32 index, Var\* value, ScriptContext\* requestContext);

static BOOL SetItem(Var instance, RecyclableObject\* object, uint64 index, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL SetItem(Var instance, RecyclableObject\* object, uint32 index, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None, BOOL skipPrototypeCheck = FALSE);

static BOOL DeleteItem(RecyclableObject\* instance, uint32 index, PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

static BOOL DeleteItem(RecyclableObject\* instance, uint64 index, PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

static Var Construct(RecyclableObject\* constructor, const Arguments args, ScriptContext\* scriptContext);

static RecyclableObject\* CreateFromConstructor(RecyclableObject\* constructor, ScriptContext\* scriptContext);

static RecyclableObject\* OrdinaryCreateFromConstructor(RecyclableObject\* constructor, RecyclableObject\* obj, DynamicObject\* intrinsicProto, ScriptContext\* scriptContext);

template<typename PropertyKeyType>

static BOOL CheckPrototypesForAccessorOrNonWritablePropertySlow(RecyclableObject\* instance, PropertyKeyType propertyKey, Var\* setterValueOrProxy, DescriptorFlags\* flags, bool isRoot, ScriptContext\* scriptContext);

static BOOL CheckPrototypesForAccessorOrNonWritableProperty(RecyclableObject\* instance, PropertyId propertyId, Var\* setterValueOrProxy, DescriptorFlags\* flags, PropertyValueInfo\* info, ScriptContext\* scriptContext);

static BOOL CheckPrototypesForAccessorOrNonWritableProperty(RecyclableObject\* instance, JavascriptString\* propertyNameString, Var\* setterValueOrProxy, DescriptorFlags\* flags, PropertyValueInfo\* info, ScriptContext\* scriptContext);

static BOOL CheckPrototypesForAccessorOrNonWritableRootProperty(RecyclableObject\* instance, PropertyId propertyId, Var\* setterValueOrProxy, DescriptorFlags\* flags, PropertyValueInfo\* info, ScriptContext\* scriptContext);

static BOOL CheckPrototypesForAccessorOrNonWritableItem(RecyclableObject\* instance, uint32 index, Var\* setterValueOrProxy, DescriptorFlags\* flags, ScriptContext\* scriptContext, BOOL skipPrototypeCheck = FALSE);

template <typename PropertyKeyType, bool unscopable>

static DescriptorFlags GetterSetter\_Impl(RecyclableObject\* instance, PropertyKeyType propertyKey, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* scriptContext);

static DescriptorFlags GetterSetterUnscopable(RecyclableObject\* instance, PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* scriptContext);

static DescriptorFlags GetterSetter(RecyclableObject\* instance, PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* scriptContext);

static DescriptorFlags GetterSetter(RecyclableObject\* instance, JavascriptString \* propertyName, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* scriptContext);

static void OP\_InvalidateProtoCaches(PropertyId propertyId, ScriptContext \*scriptContext);

static BOOL SetGlobalPropertyNoHost(wchar\_t const \* propertyName, charcount\_t propertyLength, Var value, ScriptContext \* scriptContext);

static RecyclableObject\* GetPrototype(RecyclableObject\* instance);

static RecyclableObject\* OP\_GetPrototype(Var instance, ScriptContext\* scriptContext);

static BOOL OP\_HasProperty(Var instance, PropertyId propertyId, ScriptContext\* scriptContext);

static BOOL OP\_HasOwnProperty(Var instance, PropertyId propertyId, ScriptContext\* scriptContext);

static BOOL HasOwnPropertyNoHostObject(Var instance, PropertyId propertyId);

static BOOL HasOwnPropertyNoHostObjectForHeapEnum(Var instance, PropertyId propertyId, ScriptContext\* scriptContext, Var& getter, Var& setter);

static Var GetOwnPropertyNoHostObjectForHeapEnum(Var instance, PropertyId propertyId, ScriptContext\* scriptContext, Var& getter, Var &setter);

static BOOL OP\_HasOwnPropScoped(Var instance, PropertyId propertyId, Var defaultInstance, ScriptContext\* scriptContext);

static Var OP\_GetProperty(Var instance, PropertyId propertyId, ScriptContext\* scriptContext);

static Var OP\_GetRootProperty(Var instance, PropertyId propertyId, PropertyValueInfo \* info, ScriptContext\* scriptContext);

static BOOL OP\_SetProperty(Var instance, PropertyId propertyId, Var newValue, ScriptContext\* scriptContext, PropertyValueInfo \* info = nullptr, PropertyOperationFlags flags = PropertyOperation\_None, Var thisInstance = nullptr);

static BOOL SetPropertyOnTaggedNumber(Var instance, RecyclableObject\* object, PropertyId propertyId, Var newValue, ScriptContext\* requestContext, PropertyOperationFlags flags);

static BOOL SetItemOnTaggedNumber(Var instance, RecyclableObject\* object, uint32 index, Var newValue, ScriptContext\* requestContext, PropertyOperationFlags propertyOperationFlags);

static BOOL OP\_StFunctionExpression(Var instance, PropertyId propertyId, Var newValue);

static BOOL OP\_InitProperty(Var instance, PropertyId propertyId, Var newValue);

static Var OP\_DeleteProperty(Var instance, PropertyId propertyId, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

static Var OP\_DeleteRootProperty(Var instance, PropertyId propertyId, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

static BOOL OP\_InitLetProperty(Var instance, PropertyId propertyId, Var newValue);

static BOOL OP\_InitConstProperty(Var instance, PropertyId propertyId, Var newValue);

static BOOL OP\_InitUndeclRootLetProperty(Var instance, PropertyId propertyId);

static BOOL OP\_InitUndeclRootConstProperty(Var instance, PropertyId propertyId);

static BOOL OP\_InitUndeclConsoleLetProperty(Var instance, PropertyId propertyId);

static BOOL OP\_InitUndeclConsoleConstProperty(Var instance, PropertyId propertyId);

static BOOL OP\_InitClassMember(Var instance, PropertyId propertyId, Var newValue);

static void OP\_InitClassMemberComputedName(Var object, Var elementName, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static void OP\_InitClassMemberGet(Var object, PropertyId propertyId, Var getter);

static void OP\_InitClassMemberGetComputedName(Var object, Var elementName, Var getter, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static void OP\_InitClassMemberSet(Var object, PropertyId propertyId, Var setter);

static void OP\_InitClassMemberSetComputedName(Var object, Var elementName, Var getter, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static Js::PropertyId GetPropertyId(Var propertyName, ScriptContext\* scriptContext);

static BOOL OP\_HasItem(Var instance, Var aElementIndex, ScriptContext\* scriptContext);

static Var OP\_GetElementI(Var instance, Var aElementIndex, ScriptContext\* scriptContext);

static Var OP\_GetElementI\_JIT(Var instance, Var index, ScriptContext \*scriptContext);

#if ENABLE\_NATIVE\_CODEGEN

static Var OP\_GetElementI\_JIT\_ExpectingNativeFloatArray(Var instance, Var index, ScriptContext \*scriptContext);

static Var OP\_GetElementI\_JIT\_ExpectingVarArray(Var instance, Var index, ScriptContext \*scriptContext);

#endif

static Var OP\_GetElementI\_UInt32(Var instance, uint32 aElementIndex, ScriptContext\* scriptContext);

static Var OP\_GetElementI\_UInt32\_ExpectingNativeFloatArray(Var instance, uint32 aElementIndex, ScriptContext\* scriptContext);

static Var OP\_GetElementI\_UInt32\_ExpectingVarArray(Var instance, uint32 aElementIndex, ScriptContext\* scriptContext);

static Var OP\_GetElementI\_Int32(Var instance, int32 aElementIndex, ScriptContext\* scriptContext);

static Var OP\_GetElementI\_Int32\_ExpectingNativeFloatArray(Var instance, int32 aElementIndex, ScriptContext\* scriptContext);

static Var OP\_GetElementI\_Int32\_ExpectingVarArray(Var instance, int32 aElementIndex, ScriptContext\* scriptContext);

static Var GetElementIHelper(Var instance, Var index, Var receiver, ScriptContext\* scriptContext);

static int32 OP\_GetNativeIntElementI(Var instance, Var index);

static int32 OP\_GetNativeIntElementI\_Int32(Var instance, int32 index, ScriptContext \*scriptContext);

static int32 OP\_GetNativeIntElementI\_UInt32(Var instance, uint32 index, ScriptContext \*scriptContext);

static double OP\_GetNativeFloatElementI(Var instance, Var index);

static double OP\_GetNativeFloatElementI\_Int32(Var instance, int32 index, ScriptContext \*scriptContext);

static double OP\_GetNativeFloatElementI\_UInt32(Var instance, uint32 index, ScriptContext \*scriptContext);

static Var OP\_GetMethodElement(Var instance, Var aElementIndex, ScriptContext\* scriptContext);

static Var OP\_GetMethodElement\_UInt32(Var instance, uint32 aElementIndex, ScriptContext\* scriptContext);

static Var OP\_GetMethodElement\_Int32(Var instance, int32 aElementIndex, ScriptContext\* scriptContext);

static BOOL OP\_SetElementI(Var instance, Var aElementIndex, Var aValue, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL OP\_SetElementI\_JIT(Var instance, Var aElementIndex, Var aValue, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL OP\_SetElementI\_UInt32(Var instance, uint32 aElementIndex, Var aValue, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL OP\_SetElementI\_Int32(Var instance, int aElementIndex, Var aValue, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL SetElementIHelper(Var receiver, RecyclableObject\* object, Var index, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags);

static BOOL OP\_SetNativeIntElementI(Var instance, Var aElementIndex, int32 aValue, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL OP\_SetNativeIntElementI\_UInt32(Var instance, uint32 aElementIndex, int32 aValue, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL OP\_SetNativeIntElementI\_Int32(Var instance, int aElementIndex, int32 aValue, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL OP\_SetNativeFloatElementI(Var instance, Var aElementIndex, ScriptContext\* scriptContext, PropertyOperationFlags flags, double value);

static BOOL OP\_SetNativeFloatElementI\_UInt32(Var instance, uint32 aElementIndex, ScriptContext\* scriptContext, PropertyOperationFlags flags, double value);

static BOOL OP\_SetNativeFloatElementI\_Int32(Var instance, int aElementIndex, ScriptContext\* scriptContext, PropertyOperationFlags flags, double value);

static Var OP\_DeleteElementI(Var instance, Var aElementIndex, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

static Var OP\_DeleteElementI\_UInt32(Var instance, uint32 aElementIndex, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

static Var OP\_DeleteElementI\_Int32(Var instance, int aElementIndex, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

static BOOL OP\_Memset(Var instance, int32 start, Var value, int32 length, ScriptContext\* scriptContext);

static BOOL OP\_Memcopy(Var dstInstance, int32 dstStart, Var srcInstance, int32 srcStart, int32 length, ScriptContext\* scriptContext);

static Var OP\_GetLength(Var instance, ScriptContext\* scriptContext);

static Var OP\_GetThis(Var thisVar, int moduleID, ScriptContext\* scriptContext);

static Var OP\_GetThisNoFastPath(Var thisVar, int moduleID, ScriptContext\* scriptContext);

static Var OP\_StrictGetThis(Var thisVar, ScriptContext\* scriptContext);

static bool IsThisSelf(TypeId typeId);

static Var GetThisHelper(Var thisVar, TypeId typeId, int moduleID, ScriptContext \*scriptContext);

static Var GetThisFromModuleRoot(Var thisVar);

static Var OP\_GetThisScoped(FrameDisplay \*pScope, Var defaultInstance, ScriptContext\* scriptContext);

static Var OP\_UnwrapWithObj(Var aValue);

static Var OP\_GetInstanceScoped(FrameDisplay \*pScope, PropertyId propertyId, Var rootObject, Var\* result2, ScriptContext\* scriptContext);

static BOOL OP\_InitPropertyScoped(FrameDisplay \*pScope, PropertyId propertyId, Var newValue, Var defaultInstance, ScriptContext\* scriptContext);

static BOOL OP\_InitFuncScoped(FrameDisplay \*pScope, PropertyId propertyId, Var newValue, Var defaultInstance, ScriptContext\* scriptContext);

static Var OP\_DeletePropertyScoped(

FrameDisplay \*pScope,

PropertyId propertyId,

Var defaultInstance,

ScriptContext\* scriptContext,

PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

static Var OP\_TypeofPropertyScoped(FrameDisplay \*pScope, PropertyId propertyId, Var defaultInstance, ScriptContext\* scriptContext);

static void OP\_InitGetter(Var object, PropertyId propertyId, Var getter);

static Js::PropertyId OP\_InitElemGetter(Var object, Var elementName, Var getter, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static void OP\_InitSetter(Var object, PropertyId propertyId, Var setter);

static Js::PropertyId OP\_InitElemSetter(Var object, Var elementName, Var getter, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static void OP\_InitComputedProperty(Var object, Var elementName, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static void OP\_InitProto(Var object, PropertyId propertyId, Var value);

static ForInObjectEnumerator \* OP\_GetForInEnumerator(Var enumerable, ScriptContext\* scriptContext);

static void OP\_ReleaseForInEnumerator(ForInObjectEnumerator \* enumerator, ScriptContext\* scriptContext);

static Var OP\_BrOnEmpty(ForInObjectEnumerator \* enumerator);

static BOOL OP\_BrHasSideEffects(int se,ScriptContext\* scriptContext);

static BOOL OP\_BrNotHasSideEffects(int se,ScriptContext\* scriptContext);

static BOOL OP\_BrFncEqApply(Var instance,ScriptContext\* scriptContext);

static BOOL OP\_BrFncNeqApply(Var instance,ScriptContext\* scriptContext);

static Var OP\_CmEq\_A(Js::Var a,Js::Var b,ScriptContext\* scriptContext);

static Var OP\_CmNeq\_A(Js::Var a,Js::Var b,ScriptContext\* scriptContext);

static Var OP\_CmSrEq\_A(Js::Var a,Js::Var b,ScriptContext\* scriptContext);

static Var OP\_CmSrEq\_String(Var a, Var b, ScriptContext \*scriptContext);

static Var OP\_CmSrEq\_EmptyString(Var a, ScriptContext \*scriptContext);

static Var OP\_CmSrNeq\_A(Js::Var a,Js::Var b,ScriptContext\* scriptContext);

static Var OP\_CmLt\_A(Js::Var a,Js::Var b,ScriptContext\* scriptContext);

static Var OP\_CmLe\_A(Js::Var a,Js::Var b,ScriptContext\* scriptContext);

static Var OP\_CmGt\_A(Js::Var a,Js::Var b,ScriptContext\* scriptContext);

static Var OP\_CmGe\_A(Js::Var a,Js::Var b,ScriptContext\* scriptContext);

static FunctionInfo \* JavascriptOperators::GetConstructorFunctionInfo(Var instance, ScriptContext \* scriptContext);

// Detach the type array buffer, if possible, and returns the state of the object which can be used to initialize another object

static DetachedStateBase\* DetachVarAndGetState(Var var);

static bool IsObjectDetached(Var var);

// This will return a new object from the state returned by the above operation

static Var NewVarFromDetachedState(DetachedStateBase\* state, JavascriptLibrary \*library);

static Var NewScObjectLiteral(ScriptContext\* scriptContext, const Js::PropertyIdArray \*propIds, DynamicType \*\* literalType);

static DynamicType \* EnsureObjectLiteralType(ScriptContext\* scriptContext, const Js::PropertyIdArray \*propIds, DynamicType \*\* literalType);

static uint GetLiteralSlotCapacity(Js::PropertyIdArray const \* propIds, ScriptContext \*const scriptContext);

static uint GetLiteralInlineSlotCapacity(Js::PropertyIdArray const \* propIds, ScriptContext \*const scriptContext);

static Var NewJavascriptObjectNoArg(ScriptContext\* requestContext);

static Var NewJavascriptArrayNoArg(ScriptContext\* requestContext);

static Var NewScObjectNoCtorCommon(Var instance, ScriptContext\* requestContext, bool isBaseClassConstructorNewScObject = false);

static Var NewScObjectNoCtor(Var instance, ScriptContext\* requestContext);

static Var NewScObjectNoCtorFull(Var instance, ScriptContext\* requestContext);

static Var NewScObjectNoArgNoCtorCommon(Var instance, ScriptContext\* requestContext, bool isBaseClassConstructorNewScObject = false);

static Var NewScObjectNoArgNoCtor(Var instance, ScriptContext\* requestContext);

static Var NewScObjectNoArgNoCtorFull(Var instance, ScriptContext\* requestContext);

static Var NewScObjectNoArg(Var instance, ScriptContext\* requestContext);

static Var NewScObject(const Var callee, const Arguments args, ScriptContext \*const scriptContext, const Js::AuxArray<uint32> \*spreadIndices = nullptr);

static Var AddVarsToArraySegment(SparseArraySegment<Var> \* segment, const Js::VarArray \*vars);

static void AddIntsToArraySegment(SparseArraySegment<int32> \* segment, const Js::AuxArray<int32> \*ints);

static void AddFloatsToArraySegment(SparseArraySegment<double> \* segment, const Js::AuxArray<double> \*doubles);

static void UpdateNewScObjectCache(Var function, Var instance, ScriptContext\* requestContext);

static RecyclableObject\* GetIteratorFunction(Var iterable, ScriptContext\* scriptContext);

static RecyclableObject\* GetIteratorFunction(RecyclableObject\* instance, ScriptContext \* scriptContext);

static RecyclableObject\* GetIterator(Var instance, ScriptContext\* scriptContext);

static RecyclableObject\* GetIterator(RecyclableObject\* instance, ScriptContext\* scriptContext);

static RecyclableObject\* IteratorNext(RecyclableObject\* iterator, ScriptContext\* scriptContext, Var value = nullptr);

static bool IteratorComplete(RecyclableObject\* iterResult, ScriptContext\* scriptContext);

static Var IteratorValue(RecyclableObject\* iterResult, ScriptContext\* scriptContext);

static bool IteratorStep(RecyclableObject\* iterator, ScriptContext\* scriptContext, RecyclableObject\*\* result);

static bool IteratorStepAndValue(RecyclableObject\* iterator, ScriptContext\* scriptContext, Var\* resultValue);

static void TraceUseConstructorCache(const ConstructorCache\* ctorCache, const JavascriptFunction\* ctor, bool isHit);

static void TraceUpdateConstructorCache(const ConstructorCache\* ctorCache, const FunctionBody\* ctorBody, bool updated, const wchar\_t\* reason);

static Var ConvertToUnmappedArguments(HeapArgumentsObject \*argumentsObject, uint32 paramCount, Var \*paramAddr, DynamicObject\* frameObject, Js::PropertyIdArray \*propIds, uint32 formalsCount, ScriptContext\* scriptContext);

static Js::GlobalObject \* OP\_LdRoot(ScriptContext\* scriptContext);

static Js::ModuleRoot \* GetModuleRoot(int moduleID, ScriptContext\* scriptContext);

static Js::Var OP\_LoadModuleRoot(int moduleID, ScriptContext\* scriptContext);

static Var OP\_LdNull(ScriptContext\* scriptContext);

static Var OP\_LdUndef(ScriptContext\* scriptContext);

static Var OP\_LdNaN(ScriptContext\* scriptContext);

static Var OP\_LdInfinity(ScriptContext\* scriptContext);

static FrameDisplay\* OP\_LdHandlerScope(Var argThis, ScriptContext\* scriptContext);

static FrameDisplay\* OP\_LdFrameDisplay(void \*argHead, void \*argEnv, ScriptContext\* scriptContext);

static FrameDisplay\* OP\_LdFrameDisplayNoParent(void \*argHead, ScriptContext\* scriptContext);

static FrameDisplay\* OP\_LdStrictFrameDisplay(void \*argHead, void \*argEnv, ScriptContext\* scriptContext);

static FrameDisplay\* OP\_LdStrictFrameDisplayNoParent(void \*argHead, ScriptContext\* scriptContext);

static FrameDisplay\* OP\_LdInnerFrameDisplay(void \*argHead, void \*argEnv, ScriptContext\* scriptContext);

static FrameDisplay\* OP\_LdInnerFrameDisplayNoParent(void \*argHead, ScriptContext\* scriptContext);

static FrameDisplay\* OP\_LdStrictInnerFrameDisplay(void \*argHead, void \*argEnv, ScriptContext\* scriptContext);

static FrameDisplay\* OP\_LdStrictInnerFrameDisplayNoParent(void \*argHead, ScriptContext\* scriptContext);

static void CheckInnerFrameDisplayArgument(void \*argHead);

static Var LoadHeapArguments(JavascriptFunction \*funcCallee, unsigned int count, Var \*pParams, Var frameObj, Var vArray, ScriptContext\* scriptContext, bool nonSimpleParamList);

static Var LoadHeapArgsCached(JavascriptFunction \*funcCallee, uint32 actualsCount, uint32 formalsCount, Var \*pParams, Var frameObj, ScriptContext\* scriptContext, bool nonSimpleParamList);

static HeapArgumentsObject \*CreateHeapArguments(JavascriptFunction \*funcCallee, uint32 actualsCount, uint32 formalsCount, Var frameObj, ScriptContext\* scriptContext);

static Var OP\_InitCachedScope(Var varFunc, const PropertyIdArray \*propIds, DynamicType \*\* literalType, bool formalsAreLetDecls, ScriptContext \*scriptContext);

static void OP\_InvalidateCachedScope(Var varEnv, int32 envIndex);

static void OP\_InitCachedFuncs(Var varScope, FrameDisplay \*pDisplay, const FuncInfoArray \*info, ScriptContext \*scriptContext);

static Var OP\_NewScopeObject(ScriptContext\*scriptContext);

static Var\* OP\_NewScopeSlots(unsigned int count, ScriptContext \*scriptContext, Var scope);

static Var\* OP\_NewScopeSlotsWithoutPropIds(unsigned int count, int index, ScriptContext \*scriptContext, FunctionBody \*functionBody);

static Var\* OP\_CloneScopeSlots(Var \*scopeSlots, ScriptContext \*scriptContext);

static Var OP\_NewPseudoScope(ScriptContext \*scriptContext);

static Var OP\_NewBlockScope(ScriptContext \*scriptContext);

static Var OP\_CloneBlockScope(BlockActivationObject \*blockScope, ScriptContext \*scriptContext);

static void OP\_InitClass(Var constructor, Var extends, ScriptContext \* scriptContext);

static void OP\_LoadUndefinedToElement(Var instance, PropertyId propertyId);

static void OP\_LoadUndefinedToElementDynamic(Var instance, PropertyId propertyId, ScriptContext\* scriptContext);

static void OP\_LoadUndefinedToElementScoped(FrameDisplay \*pScope, PropertyId propertyId, Var defaultInstance, ScriptContext\* scriptContext);

static Var OP\_IsInst(Var instance, Var aClass, ScriptContext\* scriptContext, IsInstInlineCache \*inlineCache);

static Var IsIn(Var argProperty, Var instance, ScriptContext\* scriptContext);

static BOOL GetRemoteTypeId(Var instance, TypeId\* typeId);

static FunctionProxy\* GetDeferredDeserializedFunctionProxy(JavascriptFunction\* func);

template <bool IsFromFullJit, class TInlineCache> static Var PatchGetValue(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template <bool IsFromFullJit, class TInlineCache> static Var PatchGetValueWithThisPtr(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var thisInstance);

template <bool IsFromFullJit, class TInlineCache> static Var PatchGetValueForTypeOf(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

static Var PatchGetValueUsingSpecifiedInlineCache(InlineCache \* inlineCache, Var instance, RecyclableObject \* object, PropertyId propertyId, ScriptContext\* scriptContext);

static Var PatchGetValueNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

static Var PatchGetValueWithThisPtrNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var thisInstance);

template <bool IsFromFullJit, class TInlineCache> static Var PatchGetRootValue(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId);

template <bool IsFromFullJit, class TInlineCache> static Var PatchGetRootValueForTypeOf(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId);

static Var PatchGetRootValueNoFastPath\_Var(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

static Var PatchGetRootValueNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId);

template <bool IsFromFullJit, class TInlineCache> static Var PatchGetPropertyScoped(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pScope, PropertyId propertyId, Var defaultInstance);

template <bool IsFromFullJit, class TInlineCache> static void PatchSetPropertyScoped(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pScope, PropertyId propertyId, Var newValue, Var defaultInstance, PropertyOperationFlags flags = PropertyOperation\_None);

template <bool IsFromFullJit, class TInlineCache> static Var PatchGetPropertyForTypeOfScoped(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pScope, PropertyId propertyId, Var defaultInstance);

template <bool IsFromFullJit, class TInlineCache> static void PatchPutValue(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var obj, PropertyId propertyId, Var newValue, PropertyOperationFlags flags = PropertyOperation\_None);

template <bool IsFromFullJit, class TInlineCache> static void PatchPutValueWithThisPtr(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var obj, PropertyId propertyId, Var newValue, Var thisInstance, PropertyOperationFlags flags = PropertyOperation\_None);

template <bool IsFromFullJit, class TInlineCache> static void PatchPutRootValue(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var obj, PropertyId propertyId, Var newValue, PropertyOperationFlags flags = PropertyOperation\_None);

template <bool IsFromFullJit, class TInlineCache> static void PatchPutValueNoLocalFastPath(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags = PropertyOperation\_None);

template <bool IsFromFullJit, class TInlineCache> static void PatchPutValueWithThisPtrNoLocalFastPath(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, Var thisInstance, PropertyOperationFlags flags = PropertyOperation\_None);

template <bool IsFromFullJit, class TInlineCache> static void PatchPutRootValueNoLocalFastPath(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags = PropertyOperation\_None);

static void PatchPutValueNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var obj, PropertyId propertyId, Var newValue, PropertyOperationFlags flags = PropertyOperation\_None);

static void PatchPutValueWithThisPtrNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var obj, PropertyId propertyId, Var newValue, Var thisInstance, PropertyOperationFlags flags = PropertyOperation\_None);

static void PatchPutRootValueNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var obj, PropertyId propertyId, Var newValue, PropertyOperationFlags flags = PropertyOperation\_None);

template <bool IsFromFullJit, class TInlineCache> static void PatchInitValue(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, RecyclableObject\* object, PropertyId propertyId, Var newValue);

static void PatchInitValueNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, RecyclableObject\* object, PropertyId propertyId, Var newValue);

template <bool IsFromFullJit, class TInlineCache> static Var PatchGetMethod(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template <bool IsFromFullJit, class TInlineCache> static Var PatchGetRootMethod(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId);

template <bool IsFromFullJit, class TInlineCache> static Var PatchScopedGetMethod(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

static Var PatchGetMethodNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

static Var PatchGetRootMethodNoFastPath\_Var(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

static Var PatchGetRootMethodNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId);

static Var PatchGetMethodFromObject(Var instance, RecyclableObject \* propertyObject, PropertyId propertyId, PropertyValueInfo \* info, ScriptContext \* scriptContext, bool isRootLd);

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

static void TracePropertyEquivalenceCheck(const JitEquivalentTypeGuard\* guard, const Type\* type, const Type\* refType, bool isEquivalent, uint failedPropertyIndex);

#endif

static bool IsStaticTypeObjTypeSpecEquivalent(const TypeEquivalenceRecord& equivalenceRecord, uint& failedIndex);

static bool IsStaticTypeObjTypeSpecEquivalent(const EquivalentPropertyEntry \*entry);

static bool CheckIfTypeIsEquivalent(Type\* type, JitEquivalentTypeGuard\* guard);

static void GetPropertyIdForInt(uint64 value, ScriptContext\* scriptContext, PropertyRecord const \*\* propertyRecord);

static void GetPropertyIdForInt(uint32 value, ScriptContext\* scriptContext, PropertyRecord const \*\* propertyRecord);

static BOOL TryConvertToUInt32(const wchar\_t\* str, int length, uint32\* value);

static BOOL ToPropertyDescriptor(Var propertySpec, PropertyDescriptor\* descriptor, ScriptContext\* scriptContext);

static Var FromPropertyDescriptor(PropertyDescriptor descriptor, ScriptContext\* scriptContext);

static void CompletePropertyDescriptor(PropertyDescriptor\* resultDescriptor, PropertyDescriptor\* likePropertyDescriptor, ScriptContext\* requestContext);

static BOOL SetPropertyDescriptor(RecyclableObject\* object, PropertyId propId, PropertyDescriptor descriptor);

static BOOL DefineOwnPropertyDescriptor(RecyclableObject\* object, PropertyId propId, const PropertyDescriptor& descriptor, bool throwOnError, ScriptContext\* scriptContext);

static BOOL DefineOwnPropertyForArray(JavascriptArray\* arr, PropertyId propId, const PropertyDescriptor& descriptor, bool throwOnError, ScriptContext\* scriptContext);

static BOOL IsCompatiblePropertyDescriptor(const PropertyDescriptor& descriptor, PropertyDescriptor\* currentDescriptor, bool isExtensible, bool throwOnError, ScriptContext\* scriptContext);

template <bool needToSetProperty>

static BOOL ValidateAndApplyPropertyDescriptor(RecyclableObject\* obj, PropertyId propId, const PropertyDescriptor& descriptor,

PropertyDescriptor\* currentPropertyDescriptor, bool isExtensible, bool throwOnError, ScriptContext\* scriptContext);

template <bool isAccessor>

static PropertyDescriptor FillMissingPropertyDescriptorFields(PropertyDescriptor descriptor, ScriptContext\* scriptContext);

static Var OP\_InvokePut(Js::ScriptContext \*scriptContext, Var function, CallInfo callInfo, ...);

static Var DefaultAccessor(RecyclableObject\* function, CallInfo callInfo, ...);

static bool IsUndefinedAccessor(Var accessor, ScriptContext\* scriptContext);

static void SetAttributes(RecyclableObject\* object, PropertyId propId, const PropertyDescriptor& descriptor, bool force);

static void OP\_ClearAttributes(Var instance, PropertyId propertyId);

static void OP\_Freeze(Var instance);

static Var RootToThisObject(const Var object, ScriptContext \* const scriptContext);

static Var CallGetter(RecyclableObject \* const function, Var const object, ScriptContext \* const scriptContext);

static void CallSetter(RecyclableObject \* const function, Var const object, Var const value, ScriptContext \* const scriptContext);

static bool CheckIfObjectAndPrototypeChainHasOnlyWritableDataProperties(RecyclableObject\* object);

static bool CheckIfPrototypeChainHasOnlyWritableDataProperties(RecyclableObject\* prototype);

static bool DoCheckIfPrototypeChainHasOnlyWritableDataProperties(RecyclableObject\* prototype);

static void OP\_SetComputedNameVar(Var method, Var computedNameVar);

static void OP\_SetHomeObj(Var method, Var homeObj);

static Var OP\_LdSuper(Var scriptFunction, ScriptContext \* scriptContext);

static Var OP\_LdSuperCtor(Var scriptFunction, ScriptContext \* scriptContext);

static Var OP\_ScopedLdSuper(Var scriptFunction, ScriptContext \* scriptContext);

static Var OP\_ScopedLdSuperCtor(Var scriptFunction, ScriptContext \* scriptContext);

static Var ScopedLdSuperHelper(Var scriptFunction, Js::PropertyId propertyId, ScriptContext \* scriptContext);

static Var OP\_ResumeYield(ResumeYieldData\* yieldData, RecyclableObject\* iterator);

static Var OP\_AsyncSpawn(Js::Var aGenerator, Js::Var aThis, ScriptContext\* scriptContext);

template <typename T>

static void \* JitRecyclerAlloc(size\_t size, Recycler\* recycler)

{

TRACK\_ALLOC\_INFO(recycler, T, Recycler, size - sizeof(T), (size\_t)-1);

return recycler->AllocZero(size);

}

static void \* AllocMemForVarArray(size\_t size, Recycler\* recycler);

static void \* AllocUninitializedNumber(RecyclerJavascriptNumberAllocator \* allocator);

static void ScriptAbort();

class EntryInfo

{

public:

static FunctionInfo DefaultAccessor;

};

template <BOOL stopAtProxy, class Func>

static void MapObjectAndPrototypes(RecyclableObject\* object, Func func);

template <BOOL stopAtProxy, class Func>

static bool MapObjectAndPrototypesUntil(RecyclableObject\* object, Func func);

#if ENABLE\_PROFILE\_INFO

static void UpdateNativeArrayProfileInfoToCreateVarArray(Var instance, const bool expectingNativeFloatArray, const bool expectingVarArray);

static bool SetElementMayHaveImplicitCalls(ScriptContext \*const scriptContext);

#endif

static RecyclableObject \*GetCallableObjectOrThrow(const Var callee, ScriptContext \*const scriptContext);

static Js::Var BoxStackInstance(Js::Var value, ScriptContext \* scriptContext, bool allowStackFunction = false);

static BOOL PropertyReferenceWalkUnscopable(Var instance, RecyclableObject\*\* propertyObject, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext);

static BOOL PropertyReferenceWalk(Var instance, RecyclableObject\*\* propertyObject, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext);

static void VarToNativeArray(Var arrayObject,

JsNativeValueType valueType,

\_\_in UINT length,

\_\_in UINT elementSize,

\_\_out\_bcount(length\*elementSize) byte\* contentBuffer,

Js::ScriptContext\* scriptContext);

static Var SpeciesConstructor(RecyclableObject\* object, Var defaultConstructor, ScriptContext\* scriptContext);

static Var GetSpecies(RecyclableObject\* constructor, ScriptContext\* scriptContext);

private:

static BOOL RelationalComparsionHelper(Var aLeft, Var aRight, ScriptContext\* scriptContext, bool leftFirst, bool undefinedAs);

template <typename ArrayType>

static void ObjectToNativeArray(ArrayType\* arrayObject,

JsNativeValueType valueType,

\_\_in UINT length,

\_\_in UINT elementSize,

\_\_out\_bcount(length\*elementSize) byte\* contentBuffer,

Js::ScriptContext\* scriptContext);

template <typename ArrayType>

static Js::Var GetElementAtIndex(ArrayType\* arrayObject, UINT index, Js::ScriptContext\* scriptContext);

#if DBG

static BOOL IsPropertyObject(RecyclableObject \* instance);

#endif

template<typename PropertyKeyType, bool doFastProtoChainCheck, bool isRoot>

static BOOL CheckPrototypesForAccessorOrNonWritablePropertyCore(RecyclableObject\* instance,

PropertyKeyType propertyKey, Var\* setterValue, DescriptorFlags\* flags, PropertyValueInfo\* info, ScriptContext\* scriptContext);

static RecyclableObject \* GetPrototypeObject(RecyclableObject \* constructorFunction, ScriptContext \* scriptContext);

static RecyclableObject \* GetPrototypeObjectForConstructorCache(RecyclableObject \* constructor, ScriptContext \* scriptContext, bool& canBeCached);

static bool PrototypeObject(Var prototypeProperty, RecyclableObject \* constructorFunction, ScriptContext \* scriptContext, RecyclableObject\*\* prototypeObject);

static Var NewScObjectHostDispatchOrProxy(RecyclableObject \* function, ScriptContext \* requestContext);

static Var NewScObjectCommon(RecyclableObject \* functionObject, FunctionInfo \* functionInfo, ScriptContext \* scriptContext, bool isBaseClassConstructorNewScObject = false);

static BOOL Reject(bool throwOnError, ScriptContext\* scriptContext, long errorCode, PropertyId propertyId);

static bool AreSamePropertyDescriptors(const PropertyDescriptor\* x, const PropertyDescriptor\* y, ScriptContext\* scriptContext);

static Var CanonicalizeAccessor(Var accessor, ScriptContext\* scriptContext);

static void BuildHandlerScope(Var argThis, RecyclableObject \* hostObject, FrameDisplay \* pScopes, ScriptContext \* scriptContext);

static void TryLoadRoot(Var& thisVar, TypeId typeId, int moduleID, ScriptContext\* scriptContext);

template <bool unscopables>

static BOOL GetProperty\_Internal(Var instance, RecyclableObject\* propertyObject, const bool isRoot, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info);

static RecyclableObject\* GetPrototypeNoTrap(RecyclableObject\* instance);

static BOOL GetPropertyReference\_Internal(Var instance, RecyclableObject\* propertyObject, const bool isRoot, PropertyId propertyId, Var\* value,ScriptContext\* requestContext, PropertyValueInfo\* info);

template <bool unscopables>

static BOOL PropertyReferenceWalk\_Impl(Var instance, RecyclableObject\*\* propertyObject, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext);

static Var TypeofFld\_Internal(Var instance, const bool isRoot, PropertyId propertyId, ScriptContext\* scriptContext);

template <bool unscopables>

static BOOL SetProperty\_Internal(Var instance, RecyclableObject\* object, const bool isRoot, PropertyId propertyId, Var newValue, PropertyValueInfo \* info, ScriptContext\* requestContext, PropertyOperationFlags flags);

template <typename TPropertyKey>

static DescriptorFlags GetRootSetter(RecyclableObject\* instance, TPropertyKey propertyKey, Var \*setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext);

static BOOL IsNumberFromNativeArray(Var instance, uint32 index, ScriptContext\* scriptContext);

static BOOL GetItemFromArrayPrototype(JavascriptArray \* arr, int32 indexInt, Var \* result, ScriptContext \* scriptContext);

template <typename T>

static BOOL OP\_GetElementI\_ArrayFastPath(T \* arr, int indexInt, Var \* result, ScriptContext \* scriptContext);

static ImplicitCallFlags CacheAndClearImplicitBit(ScriptContext\* scriptContext);

static ImplicitCallFlags CheckAndUpdateFunctionBodyWithImplicitFlag(FunctionBody\* functionBody);

static void RestoreImplicitFlag(ScriptContext\* scriptContext, ImplicitCallFlags prevImplicitCallFlags, ImplicitCallFlags currImplicitCallFlags);

static BOOL ToPropertyDescriptorForProxyObjects(Var propertySpec, PropertyDescriptor\* descriptor, ScriptContext\* scriptContext);

static BOOL ToPropertyDescriptorForGenericObjects(Var propertySpec, PropertyDescriptor\* descriptor, ScriptContext\* scriptContext);

};

} // namespace Js

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#include "Types\PathTypeHandler.h"

#include "Types\PropertyIndexRanges.h"

#include "Types\WithScopeObject.h"

#include "Types\SpreadArgument.h"

#include "Library\JavascriptPromise.h"

#include "Library\JavascriptRegularExpression.h"

#include "Library\ThrowErrorObject.h"

#include "Library\JavascriptGeneratorFunction.h"

#include "Types\DynamicObjectEnumerator.h"

#include "Types\DynamicObjectSnapshotEnumerator.h"

#include "Types\DynamicObjectSnapshotEnumeratorWPCache.h"

#include "Library\ForInObjectEnumerator.h"

#include "Library\ES5Array.h"

#ifndef SCRIPT\_DIRECT\_TYPE

typedef enum JsNativeValueType

{

JsInt8Type,

JsUint8Type,

JsInt16Type,

JsUint16Type,

JsInt32Type,

JsUint32Type,

JsInt64Type,

JsUint64Type,

JsFloatType,

JsDoubleType,

JsNativeStringType

} JsNativeValueType;

typedef struct JsNativeString

{

unsigned int length;

LPCWSTR str;

} JsNativeString;

#endif

namespace Js

{

DEFINE\_RECYCLER\_TRACKER\_ARRAY\_PERF\_COUNTER(Var);

DEFINE\_RECYCLER\_TRACKER\_PERF\_COUNTER(FrameDisplay);

enum IndexType

{

IndexType\_Number,

IndexType\_PropertyId,

IndexType\_JavascriptString

};

IndexType GetIndexTypeFromString(wchar\_t const \* propertyName, charcount\_t propertyLength, ScriptContext\* scriptContext, uint32\* index, PropertyRecord const\*\* propertyRecord, bool createIfNotFound)

{

if (JavascriptOperators::TryConvertToUInt32(propertyName, propertyLength, index) &&

(\*index != JavascriptArray::InvalidIndex))

{

return IndexType\_Number;

}

else

{

if (createIfNotFound)

{

scriptContext->GetOrAddPropertyRecord(propertyName, propertyLength, propertyRecord);

}

else

{

scriptContext->FindPropertyRecord(propertyName, propertyLength, propertyRecord);

}

return IndexType\_PropertyId;

}

}

IndexType GetIndexType(Var indexVar, ScriptContext\* scriptContext, uint32\* index, PropertyRecord const \*\* propertyRecord, JavascriptString \*\* propertyNameString, bool createIfNotFound, bool preferJavascriptStringOverPropertyRecord)

{

indexVar = JavascriptConversion::ToPrimitive(indexVar, JavascriptHint::HintString, scriptContext);

// CONSIDER: Only OP\_SetElementI and OP\_GetElementI use and take advantage of the

// IndexType\_JavascriptString result. Consider modifying other callers of GetIndexType to take

// advantage of non-interned property strings where appropriate.

if (TaggedInt::Is(indexVar))

{

int indexInt = TaggedInt::ToInt32(indexVar);

if (indexInt >= 0)

{

\*index = (uint)indexInt;

return IndexType\_Number;

}

else

{

wchar\_t buffer[20];

::\_itow\_s(indexInt, buffer, sizeof(buffer)/sizeof(wchar\_t), 10);

charcount\_t length = JavascriptString::GetBufferLength(buffer);

if (createIfNotFound || preferJavascriptStringOverPropertyRecord)

{

// When preferring JavascriptString objects, just return a PropertyRecord instead

// of creating temporary JavascriptString objects for every negative integer that

// comes through here.

scriptContext->GetOrAddPropertyRecord(buffer, length, propertyRecord);

}

else

{

scriptContext->FindPropertyRecord(buffer, length, propertyRecord);

}

return IndexType\_PropertyId;

}

}

else if (JavascriptSymbol::Is(indexVar))

{

JavascriptSymbol\* symbol = JavascriptSymbol::FromVar(indexVar);

// JavascriptSymbols cannot add a new PropertyRecord - they correspond to one and only one existing PropertyRecord.

// We already know what the PropertyRecord is since it is stored in the JavascriptSymbol itself so just return it.

\*propertyRecord = symbol->GetValue();

return IndexType\_PropertyId;

}

else

{

JavascriptString\* indexStr = JavascriptConversion::ToString(indexVar, scriptContext);

wchar\_t const \* propertyName = indexStr->GetString();

charcount\_t const propertyLength = indexStr->GetLength();

if (!createIfNotFound && preferJavascriptStringOverPropertyRecord)

{

if (JavascriptOperators::TryConvertToUInt32(propertyName, propertyLength, index) &&

(\*index != JavascriptArray::InvalidIndex))

{

return IndexType\_Number;

}

\*propertyNameString = indexStr;

return IndexType\_JavascriptString;

}

return GetIndexTypeFromString(propertyName, propertyLength, scriptContext, index, propertyRecord, createIfNotFound);

}

}

IndexType GetIndexType(Var indexVar, ScriptContext\* scriptContext, uint32\* index, PropertyRecord const \*\* propertyRecord, bool createIfNotFound)

{

return GetIndexType(indexVar, scriptContext, index, propertyRecord, nullptr, createIfNotFound, false);

}

BOOL FEqualDbl(double dbl1, double dbl2)

{

// If the low ulongs don't match, they can't be equal.

if (Js::NumberUtilities::LuLoDbl(dbl1) != Js::NumberUtilities::LuLoDbl(dbl2))

return FALSE;

// If the high ulongs don't match, they can be equal iff one is -0 and

// the other is +0.

if (Js::NumberUtilities::LuHiDbl(dbl1) != Js::NumberUtilities::LuHiDbl(dbl2))

{

return 0x80000000 == (Js::NumberUtilities::LuHiDbl(dbl1) | Js::NumberUtilities::LuHiDbl(dbl2)) &&

0 == Js::NumberUtilities::LuLoDbl(dbl1);

}

// The bit patterns match. They are equal iff they are not Nan.

return !Js::NumberUtilities::IsNan(dbl1);

}

Var JavascriptOperators::OP\_ApplyArgs(Var func, Var instance, \_\_in\_xcount(8) void\*\* stackPtr, CallInfo callInfo, ScriptContext\* scriptContext)

{

int argCount=callInfo.Count;

///

/// Check func has internal [[Call]] property

/// If not, throw TypeError

///

if (!JavascriptConversion::IsCallable(func)) {

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedFunction);

}

// Fix callInfo: expect result/value, and none of other flags are currently applicable.

// OP\_ApplyArgs expects a result. Neither of {jit, interpreted} mode sends correct callFlags:

// LdArgCnt -- jit sends whatever was passed to current function, interpreter always sends 0.

// See Win8 bug 490489.

callInfo.Flags = CallFlags\_Value;

RecyclableObject \*funcPtr = RecyclableObject::FromVar(func);

PROBE\_STACK(scriptContext, Js::Constants::MinStackDefault+argCount\*4);

JavascriptMethod entryPoint=funcPtr->GetEntryPoint();

Var ret;

switch (argCount) {

case 0:

Assert(false);

ret=entryPoint(funcPtr,callInfo);

break;

case 1:

ret=entryPoint(funcPtr,callInfo,instance);

break;

case 2:

ret=entryPoint(funcPtr,callInfo,instance,stackPtr[0]);

break;

case 3:

ret=entryPoint(funcPtr,callInfo,instance,stackPtr[0],stackPtr[1]);

break;

case 4:

ret=entryPoint(funcPtr,callInfo,instance,stackPtr[0],stackPtr[1],stackPtr[2]);

break;

case 5:

ret=entryPoint(funcPtr,callInfo,instance,stackPtr[0],stackPtr[1],stackPtr[2],stackPtr[3]);

break;

case 6:

ret=entryPoint(funcPtr,callInfo,instance,stackPtr[0],stackPtr[1],stackPtr[2],stackPtr[3],stackPtr[4]);

break;

case 7:

ret=entryPoint(funcPtr,callInfo,instance,stackPtr[0],stackPtr[1],stackPtr[2],stackPtr[3],stackPtr[4],stackPtr[5]);

break;

default: {

// Don't need stack probe here- we just did so above

Arguments args(callInfo,stackPtr-1);

ret=JavascriptFunction::CallFunction<false>(funcPtr,entryPoint,args);

}

break;

}

return ret;

}

#ifdef \_M\_IX86

// Alias for overloaded JavascriptNumber::ToVar so it can be called unambiguously from native code

Var JavascriptOperators::Int32ToVar(int32 value, ScriptContext\* scriptContext)

{

return JavascriptNumber::ToVar(value, scriptContext);

}

// Alias for overloaded JavascriptNumber::ToVar so it can be called unambiguously from native code

Var JavascriptOperators::Int32ToVarInPlace(int32 value, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

return JavascriptNumber::ToVarInPlace(value, scriptContext, result);

}

// Alias for overloaded JavascriptNumber::ToVar so it can be called unambiguously from native code

Var JavascriptOperators::UInt32ToVar(uint32 value, ScriptContext\* scriptContext)

{

return JavascriptNumber::ToVar(value, scriptContext);

}

// Alias for overloaded JavascriptNumber::ToVar so it can be called unambiguously from native code

Var JavascriptOperators::UInt32ToVarInPlace(uint32 value, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

return JavascriptNumber::ToVarInPlace(value, scriptContext, result);

}

#endif

Var JavascriptOperators::OP\_FinishOddDivBy2(uint32 value, ScriptContext \*scriptContext)

{

return JavascriptNumber::New((double)(value + 0.5), scriptContext);

}

Var JavascriptOperators::ToNumberInPlace(Var aRight, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

if (TaggedInt::Is(aRight) || JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return aRight;

}

return JavascriptNumber::ToVarInPlace(JavascriptConversion::ToNumber(aRight, scriptContext), scriptContext, result);

}

Var JavascriptOperators::Typeof(Var var, ScriptContext\* scriptContext)

{

switch (JavascriptOperators::GetTypeId(var))

{

case TypeIds\_Undefined:

return scriptContext->GetLibrary()->GetUndefinedDisplayString();

case TypeIds\_Null:

//null

return scriptContext->GetLibrary()->GetObjectTypeDisplayString();

case TypeIds\_Integer:

case TypeIds\_Number:

case TypeIds\_Int64Number:

case TypeIds\_UInt64Number:

return scriptContext->GetLibrary()->GetNumberTypeDisplayString();

case TypeIds\_SIMDFloat32x4:

if (scriptContext->GetConfig()->IsSimdjsEnabled())

{

return scriptContext->GetLibrary()->GetSIMDFloat32x4DisplayString();

}

case TypeIds\_SIMDFloat64x2:

if (scriptContext->GetConfig()->IsSimdjsEnabled())

{

return scriptContext->GetLibrary()->GetSIMDFloat64x2DisplayString();

}

case TypeIds\_SIMDInt32x4:

if (scriptContext->GetConfig()->IsSimdjsEnabled())

{

return scriptContext->GetLibrary()->GetSIMDInt32x4DisplayString();

}

case TypeIds\_SIMDInt8x16:

if (scriptContext->GetConfig()->IsSimdjsEnabled())

{

return scriptContext->GetLibrary()->GetSIMDInt8x16DisplayString();

}

default:

// Falsy objects are typeof 'undefined'.

if (RecyclableObject::FromVar(var)->GetType()->IsFalsy())

{

return scriptContext->GetLibrary()->GetUndefinedDisplayString();

}

else

{

return RecyclableObject::FromVar(var)->GetTypeOfString(scriptContext);

}

}

}

Var JavascriptOperators::TypeofFld(Var instance, PropertyId propertyId, ScriptContext\* scriptContext)

{

return TypeofFld\_Internal(instance, false, propertyId, scriptContext);

}

Var JavascriptOperators::TypeofRootFld(Var instance, PropertyId propertyId, ScriptContext\* scriptContext)

{

return TypeofFld\_Internal(instance, true, propertyId, scriptContext);

}

Var JavascriptOperators::TypeofFld\_Internal(Var instance, const bool isRoot, PropertyId propertyId, ScriptContext\* scriptContext)

{

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined , scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

Var value;

try

{

Js::JavascriptExceptionOperators::AutoCatchHandlerExists autoCatchHandlerExists(scriptContext);

// In edge mode, spec compat is more important than backward compat. Use spec/web behavior here

if (isRoot

? !JavascriptOperators::GetRootProperty(instance, propertyId, &value, scriptContext)

: !JavascriptOperators::GetProperty(instance, object, propertyId, &value, scriptContext))

{

return scriptContext->GetLibrary()->GetUndefinedDisplayString();

}

if (!scriptContext->IsUndeclBlockVar(value))

{

return JavascriptOperators::Typeof(value, scriptContext);

}

}

catch(Js::JavascriptExceptionObject \* )

{

return scriptContext->GetLibrary()->GetUndefinedDisplayString();

}

Assert(scriptContext->IsUndeclBlockVar(value));

Assert(scriptContext->GetConfig()->IsLetAndConstEnabled());

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UseBeforeDeclaration);

}

Var JavascriptOperators::TypeofElem\_UInt32(Var instance, uint32 index, ScriptContext\* scriptContext)

{

if (JavascriptOperators::IsNumberFromNativeArray(instance, index, scriptContext))

return scriptContext->GetLibrary()->GetNumberTypeDisplayString();

#if FLOATVAR

return TypeofElem(instance, Js::JavascriptNumber::ToVar(index, scriptContext), scriptContext);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return TypeofElem(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer), scriptContext);

#endif

}

Var JavascriptOperators::TypeofElem\_Int32(Var instance, int32 index, ScriptContext\* scriptContext)

{

if (JavascriptOperators::IsNumberFromNativeArray(instance, index, scriptContext))

return scriptContext->GetLibrary()->GetNumberTypeDisplayString();

#if FLOATVAR

return TypeofElem(instance, Js::JavascriptNumber::ToVar(index, scriptContext), scriptContext);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return TypeofElem(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer), scriptContext);

#endif

}

Js::JavascriptString\* GetPropertyDisplayNameForError(Var prop, ScriptContext\* scriptContext)

{

JavascriptString\* str;

if (JavascriptSymbol::Is(prop))

{

str = JavascriptSymbol::ToString(JavascriptSymbol::FromVar(prop)->GetValue(), scriptContext);

}

else

{

str = JavascriptConversion::ToString(prop, scriptContext);

}

return str;

}

Var JavascriptOperators::TypeofElem(Var instance, Var index, ScriptContext\* scriptContext)

{

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined, GetPropertyDisplayNameForError(index, scriptContext));

}

Var member;

uint32 indexVal;

PropertyRecord const \* propertyRecord = nullptr;

ThreadContext\* threadContext = scriptContext->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

try

{

Js::JavascriptExceptionOperators::AutoCatchHandlerExists autoCatchHandlerExists(scriptContext);

// For JS Objects, don't create the propertyId if not already added

bool createIfNotFound = !IsJsNativeObject(object) ||

(DynamicType::Is(object->GetTypeId()) && static\_cast<DynamicObject\*>(object)->GetTypeHandler()->IsStringTypeHandler()) || JavascriptProxy::Is(object);

if (GetIndexType(index, scriptContext, &indexVal, &propertyRecord, createIfNotFound) == IndexType\_Number)

{

// In edge mode, we don't need to worry about the special "unknown" behavior. If the item is not available from Get,

// just return undefined.

if (!JavascriptOperators::GetItem(instance, object, indexVal, &member, scriptContext))

{

// If the instance doesn't have the item, typeof result is "undefined".

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

return scriptContext->GetLibrary()->GetUndefinedDisplayString();

}

}

else if (propertyRecord == nullptr)

{

Assert(IsJsNativeObject(object));

#if DBG

JavascriptString\* indexStr = JavascriptConversion::ToString(index, scriptContext);

PropertyRecord const \* debugPropertyRecord;

scriptContext->GetOrAddPropertyRecord(indexStr->GetString(), indexStr->GetLength(), &debugPropertyRecord);

AssertMsg(!JavascriptOperators::GetProperty(instance, object, debugPropertyRecord->GetPropertyId(), &member, scriptContext), "how did this property come? See OS Bug 2727708 if you see this come from the web");

#endif

// If the instance doesn't have the property, typeof result is "undefined".

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

return scriptContext->GetLibrary()->GetUndefinedDisplayString();

}

else

{

if (!JavascriptOperators::GetProperty(instance, object, propertyRecord->GetPropertyId(), &member, scriptContext))

{

// If the instance doesn't have the property, typeof result is "undefined".

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

return scriptContext->GetLibrary()->GetUndefinedDisplayString();

}

}

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

return JavascriptOperators::Typeof(member, scriptContext);

}

catch(Js::JavascriptExceptionObject \* )

{

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

return scriptContext->GetLibrary()->GetUndefinedDisplayString();

}

}

//

// Delete the given Var

//

Var JavascriptOperators::Delete(Var var, ScriptContext\* scriptContext)

{

return scriptContext->GetLibrary()->GetTrue();

}

BOOL JavascriptOperators::Equal\_Full(Var aLeft, Var aRight, ScriptContext\* requestContext)

{

//

// Fast-path SmInts and paired Number combinations.

//

if (aLeft == aRight)

{

if (JavascriptNumber::Is(aLeft) && JavascriptNumber::IsNan(JavascriptNumber::GetValue(aLeft)))

{

return false;

}

else if (JavascriptVariantDate::Is(aLeft) == false) // only need to check on aLeft - since they are the same var, aRight would do the same

{

return true;

}

else

{

//In ES5 mode strict equals (===) on same instance of object type VariantDate succeeds.

//Hence equals needs to succeed.

return true;

}

}

BOOL result = false;

if (TaggedInt::Is(aLeft))

{

if (TaggedInt::Is(aRight))

{

// If aLeft == aRight, we would already have returned true above.

return false;

}

else if (JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return TaggedInt::ToDouble(aLeft) == JavascriptNumber::GetValue(aRight);

}

else

{

BOOL res = RecyclableObject::FromVar(aRight)->Equals(aLeft, &result, requestContext);

AssertMsg(res, "Should have handled this");

return result;

}

}

else if (JavascriptNumber::Is\_NoTaggedIntCheck(aLeft))

{

if (TaggedInt::Is(aRight))

{

return TaggedInt::ToDouble(aRight) == JavascriptNumber::GetValue(aLeft);

}

else if(JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return JavascriptNumber::GetValue(aLeft) == JavascriptNumber::GetValue(aRight);

}

else

{

BOOL res = RecyclableObject::FromVar(aRight)->Equals(aLeft, &result, requestContext);

AssertMsg(res, "Should have handled this");

return result;

}

}

if (RecyclableObject::FromVar(aLeft)->Equals(aRight, &result, requestContext))

{

return result;

}

else

{

return false;

}

}

BOOL JavascriptOperators::Greater\_Full(Var aLeft,Var aRight,ScriptContext\* scriptContext)

{

return RelationalComparsionHelper(aRight, aLeft, scriptContext, false, false);

}

BOOL JavascriptOperators::Less\_Full(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

return RelationalComparsionHelper(aLeft, aRight, scriptContext, true, false);

}

BOOL JavascriptOperators::RelationalComparsionHelper(Var aLeft, Var aRight, ScriptContext\* scriptContext, bool leftFirst, bool undefinedAs)

{

TypeId typeId = JavascriptOperators::GetTypeId(aLeft);

if (typeId == TypeIds\_Null)

{

aLeft=TaggedInt::ToVarUnchecked(0);

}

else if (typeId == TypeIds\_Undefined)

{

aLeft=scriptContext->GetLibrary()->GetNaN();

}

typeId = JavascriptOperators::GetTypeId(aRight);

if (typeId == TypeIds\_Null)

{

aRight=TaggedInt::ToVarUnchecked(0);

}

else if (typeId == TypeIds\_Undefined)

{

aRight=scriptContext->GetLibrary()->GetNaN();

}

double dblLeft, dblRight;

Redo:

TypeId leftType = JavascriptOperators::GetTypeId(aLeft);

TypeId rightType = JavascriptOperators::GetTypeId(aRight);

switch (leftType)

{

case TypeIds\_Integer:

dblLeft = TaggedInt::ToDouble(aLeft);

switch (rightType)

{

case TypeIds\_Integer:

dblRight = TaggedInt::ToDouble(aRight);

break;

case TypeIds\_Number:

dblRight = JavascriptNumber::GetValue(aRight);

break;

default:

dblRight = JavascriptConversion::ToNumber(aRight, scriptContext);

break;

}

break;

case TypeIds\_Number:

dblLeft = JavascriptNumber::GetValue(aLeft);

switch (rightType)

{

case TypeIds\_Integer:

dblRight = TaggedInt::ToDouble(aRight);

break;

case TypeIds\_Number:

dblRight = JavascriptNumber::GetValue(aRight);

break;

default:

dblRight = JavascriptConversion::ToNumber(aRight, scriptContext);

break;

}

break;

case TypeIds\_Int64Number:

{

switch (rightType)

{

case TypeIds\_Int64Number:

{

\_\_int64 leftValue = JavascriptInt64Number::FromVar(aLeft)->GetValue();

\_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return leftValue < rightValue;

}

break;

case TypeIds\_UInt64Number:

{

\_\_int64 leftValue = JavascriptInt64Number::FromVar(aLeft)->GetValue();

unsigned \_\_int64 rightValue = JavascriptUInt64Number::FromVar(aRight)->GetValue();

if (rightValue <= INT\_MAX && leftValue >= 0)

{

return leftValue < (\_\_int64)rightValue;

}

}

break;

}

dblLeft = (double)JavascriptInt64Number::FromVar(aLeft)->GetValue();

dblRight = JavascriptConversion::ToNumber(aRight, scriptContext);

}

break;

// we cannot do double conversion between 2 int64 numbers as we can get wrong result after conversion

// i.e., two different numbers become the same after losing precision. We'll continue dbl comparison

// if either number is not an int64 number.

case TypeIds\_UInt64Number:

{

switch (rightType)

{

case TypeIds\_Int64Number:

{

unsigned \_\_int64 leftValue = JavascriptUInt64Number::FromVar(aLeft)->GetValue();

\_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

if (leftValue < INT\_MAX && rightValue >= 0)

{

return (\_\_int64)leftValue < rightValue;

}

}

break;

case TypeIds\_UInt64Number:

{

unsigned \_\_int64 leftValue = JavascriptUInt64Number::FromVar(aLeft)->GetValue();

unsigned \_\_int64 rightValue = JavascriptUInt64Number::FromVar(aRight)->GetValue();

return leftValue < rightValue;

}

break;

}

dblLeft = (double)JavascriptUInt64Number::FromVar(aLeft)->GetValue();

dblRight = JavascriptConversion::ToNumber(aRight, scriptContext);

}

break;

case TypeIds\_String:

switch (rightType)

{

case TypeIds\_Integer:

case TypeIds\_Number:

case TypeIds\_Boolean:

break;

default:

aRight = JavascriptConversion::ToPrimitive(aRight, JavascriptHint::HintNumber, scriptContext);

rightType = JavascriptOperators::GetTypeId(aRight);

if (rightType != TypeIds\_String)

{

dblRight = JavascriptConversion::ToNumber(aRight, scriptContext);

break;

}

case TypeIds\_String:

return JavascriptString::LessThan(aLeft, aRight);

}

dblLeft = JavascriptConversion::ToNumber(aLeft, scriptContext);

dblRight = JavascriptConversion::ToNumber(aRight, scriptContext);

break;

case TypeIds\_Boolean:

dblLeft = JavascriptConversion::ToNumber(aLeft, scriptContext);

dblRight = JavascriptConversion::ToNumber(aRight, scriptContext);

break;

default:

if (leftFirst)

{

aLeft = JavascriptConversion::ToPrimitive(aLeft, JavascriptHint::HintNumber, scriptContext);

aRight = JavascriptConversion::ToPrimitive(aRight, JavascriptHint::HintNumber, scriptContext);

}

else

{

aRight = JavascriptConversion::ToPrimitive(aRight, JavascriptHint::HintNumber, scriptContext);

aLeft = JavascriptConversion::ToPrimitive(aLeft, JavascriptHint::HintNumber, scriptContext);

}

goto Redo;

}

//

// And +0,-0 that is not implemented fully

//

if (JavascriptNumber::IsNan(dblLeft) || JavascriptNumber::IsNan(dblRight))

{

return undefinedAs;

}

// this will succeed for -0.0 == 0.0 case as well

if (dblLeft == dblRight)

{

return false;

}

return dblLeft < dblRight;

}

BOOL JavascriptOperators::StrictEqualString(Var aLeft, Var aRight)

{

Assert(JavascriptOperators::GetTypeId(aRight) == TypeIds\_String);

if (JavascriptOperators::GetTypeId(aLeft) != TypeIds\_String)

return false;

return JavascriptString::Equals(aLeft, aRight);

}

BOOL JavascriptOperators::StrictEqualEmptyString(Var aLeft)

{

TypeId leftType = JavascriptOperators::GetTypeId(aLeft);

if (leftType != TypeIds\_String)

return false;

return JavascriptString::FromVar(aLeft)->GetLength() == 0;

}

BOOL JavascriptOperators::StrictEqual(Var aLeft, Var aRight, ScriptContext\* requestContext)

{

double dblLeft, dblRight;

TypeId leftType = JavascriptOperators::GetTypeId(aLeft);

TypeId rightType = JavascriptOperators::GetTypeId(aRight);

switch (leftType)

{

case TypeIds\_String:

switch (rightType)

{

case TypeIds\_String:

return JavascriptString::Equals(aLeft, aRight);

}

return FALSE;

case TypeIds\_Integer:

switch (rightType)

{

case TypeIds\_Integer:

return aLeft == aRight;

// we don't need to worry about int64: it cannot equal as we create

// JavascriptInt64Number only in overflow scenarios.

case TypeIds\_Number:

dblLeft = TaggedInt::ToDouble(aLeft);

dblRight = JavascriptNumber::GetValue(aRight);

goto CommonNumber;

}

return FALSE;

case TypeIds\_Int64Number:

switch (rightType)

{

case TypeIds\_Int64Number:

{

\_\_int64 leftValue = JavascriptInt64Number::FromVar(aLeft)->GetValue();

\_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return leftValue == rightValue;

}

case TypeIds\_UInt64Number:

{

\_\_int64 leftValue = JavascriptInt64Number::FromVar(aLeft)->GetValue();

unsigned \_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return ((unsigned \_\_int64)leftValue == rightValue);

}

case TypeIds\_Number:

dblLeft = (double)JavascriptInt64Number::FromVar(aLeft)->GetValue();

dblRight = JavascriptNumber::GetValue(aRight);

goto CommonNumber;

}

return FALSE;

case TypeIds\_UInt64Number:

switch (rightType)

{

case TypeIds\_Int64Number:

{

unsigned \_\_int64 leftValue = JavascriptUInt64Number::FromVar(aLeft)->GetValue();

\_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return (leftValue == (unsigned \_\_int64)rightValue);

}

case TypeIds\_UInt64Number:

{

unsigned \_\_int64 leftValue = JavascriptUInt64Number::FromVar(aLeft)->GetValue();

unsigned \_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return leftValue == rightValue;

}

case TypeIds\_Number:

dblLeft = (double)JavascriptUInt64Number::FromVar(aLeft)->GetValue();

dblRight = JavascriptNumber::GetValue(aRight);

goto CommonNumber;

}

return FALSE;

case TypeIds\_Number:

switch (rightType)

{

case TypeIds\_Integer:

dblLeft = JavascriptNumber::GetValue(aLeft);

dblRight = TaggedInt::ToDouble(aRight);

goto CommonNumber;

case TypeIds\_Int64Number:

dblLeft = JavascriptNumber::GetValue(aLeft);

dblRight = (double)JavascriptInt64Number::FromVar(aRight)->GetValue();

goto CommonNumber;

case TypeIds\_UInt64Number:

dblLeft = JavascriptNumber::GetValue(aLeft);

dblRight = (double)JavascriptUInt64Number::FromVar(aRight)->GetValue();

goto CommonNumber;

case TypeIds\_Number:

dblLeft = JavascriptNumber::GetValue(aLeft);

dblRight = JavascriptNumber::GetValue(aRight);

CommonNumber:

return FEqualDbl(dblLeft, dblRight);

}

return FALSE;

case TypeIds\_Boolean:

switch (rightType)

{

case TypeIds\_Boolean:

return aLeft == aRight;

}

return FALSE;

case TypeIds\_Undefined:

return rightType == TypeIds\_Undefined;

case TypeIds\_Null:

return rightType == TypeIds\_Null;

case TypeIds\_Array:

return (rightType == TypeIds\_Array && aLeft == aRight);

case TypeIds\_Symbol:

switch (rightType)

{

case TypeIds\_Symbol:

{

const PropertyRecord\* leftValue = JavascriptSymbol::FromVar(aLeft)->GetValue();

const PropertyRecord\* rightValue = JavascriptSymbol::FromVar(aRight)->GetValue();

return leftValue == rightValue;

}

}

return false;

case TypeIds\_GlobalObject:

case TypeIds\_HostDispatch:

switch (rightType)

{

case TypeIds\_HostDispatch:

case TypeIds\_GlobalObject:

{

BOOL result;

if(RecyclableObject::FromVar(aLeft)->StrictEquals(aRight, &result, requestContext))

{

return result;

}

return false;

}

}

break;

case TypeIds\_Function:

if (rightType == TypeIds\_Function)

{

// In ES5 in certain cases (ES5 10.6.14(strict), 13.2.19(strict), 15.3.4.5.20-21) we return a function that throws type error.

// For different scenarios we return different instances of the function, which differ by exception/error message.

// According to ES5, this is the same [[ThrowTypeError]] (thrower) internal function, thus they should be equal.

if (JavascriptFunction::FromVar(aLeft)->IsThrowTypeErrorFunction() &&

JavascriptFunction::FromVar(aRight)->IsThrowTypeErrorFunction())

{

return true;

}

}

break;

}

if (RecyclableObject::FromVar(aLeft)->CanHaveInterceptors())

{

BOOL result;

if (RecyclableObject::FromVar(aLeft)->StrictEquals(aRight, &result, requestContext))

{

if (result)

{

return TRUE;

}

}

}

if (!TaggedNumber::Is(aRight) && RecyclableObject::FromVar(aRight)->CanHaveInterceptors())

{

BOOL result;

if (RecyclableObject::FromVar(aRight)->StrictEquals(aLeft, &result, requestContext))

{

if (result)

{

return TRUE;

}

}

}

return aLeft == aRight;

}

BOOL JavascriptOperators::HasOwnProperty(Var instance, PropertyId propertyId, ScriptContext \*requestContext)

{

BOOL result;

if (TaggedNumber::Is(instance))

{

result = false;

}

else

{

RecyclableObject\* object = RecyclableObject::FromVar(instance);

if (JavascriptProxy::Is(instance))

{

PropertyDescriptor desc;

return GetOwnPropertyDescriptor(object, propertyId, requestContext, &desc);

}

else

{

return object && object->HasOwnProperty(propertyId);

}

}

return result;

}

BOOL JavascriptOperators::GetOwnAccessors(Var instance, PropertyId propertyId, Var\* getter, Var\* setter, ScriptContext \* requestContext)

{

BOOL result;

if (TaggedNumber::Is(instance))

{

result = false;

}

else

{

RecyclableObject\* object = RecyclableObject::FromVar(instance);

result = object && object->GetAccessors(propertyId, getter, setter, requestContext);

}

return result;

}

Var JavascriptOperators::GetOwnPropertyNames(Var instance, ScriptContext \*scriptContext)

{

RecyclableObject \*object = RecyclableObject::FromVar(ToObject(instance, scriptContext));

if (JavascriptProxy::Is(instance))

{

JavascriptProxy\* proxy = JavascriptProxy::FromVar(instance);

return proxy->PropertyKeysTrap(JavascriptProxy::KeysTrapKind::GetOwnPropertyNamesKind);

}

return JavascriptObject::CreateOwnStringPropertiesHelper(object, scriptContext);

}

Var JavascriptOperators::GetOwnPropertySymbols(Var instance, ScriptContext \*scriptContext)

{

RecyclableObject \*object = RecyclableObject::FromVar(ToObject(instance, scriptContext));

CHAKRATEL\_LANGSTATS\_INC\_BUILTINCOUNT(GetOwnPropertySymbolsCount);

if (JavascriptProxy::Is(instance))

{

JavascriptProxy\* proxy = JavascriptProxy::FromVar(instance);

return proxy->PropertyKeysTrap(JavascriptProxy::KeysTrapKind::GetOwnPropertySymbolKind);

}

return JavascriptObject::CreateOwnSymbolPropertiesHelper(object, scriptContext);

}

Var JavascriptOperators::GetOwnPropertyKeys(Var instance, ScriptContext\* scriptContext)

{

RecyclableObject \*object = RecyclableObject::FromVar(ToObject(instance, scriptContext));

if (JavascriptProxy::Is(instance))

{

JavascriptProxy\* proxy = JavascriptProxy::FromVar(instance);

return proxy->PropertyKeysTrap(JavascriptProxy::KeysTrapKind::KeysKind);

}

return JavascriptObject::CreateOwnStringSymbolPropertiesHelper(object, scriptContext);

}

Var JavascriptOperators::GetOwnEnumerablePropertyNames(Var instance, ScriptContext\* scriptContext)

{

RecyclableObject \*object = RecyclableObject::FromVar(ToObject(instance, scriptContext));

if (JavascriptProxy::Is(instance))

{

JavascriptProxy\* proxy = JavascriptProxy::FromVar(instance);

Var result = proxy->PropertyKeysTrap(JavascriptProxy::KeysTrapKind::GetOwnPropertyNamesKind);

AssertMsg(JavascriptArray::Is(result), "PropertyKeysTrap should return JavascriptArray.");

JavascriptArray\* proxyResult;

JavascriptArray\* proxyResultToReturn = scriptContext->GetLibrary()->CreateArray(0);

if (JavascriptArray::Is(result))

{

proxyResult = JavascriptArray::FromVar(result);

}

else

{

return proxyResultToReturn;

}

// filter enumerable keys

uint32 resultLength = proxyResult->GetLength();

Var element;

const Js::PropertyRecord \*propertyRecord;

uint32 index = 0;

for (uint32 i = 0; i < resultLength; i++)

{

element = proxyResult->DirectGetItem(i);

Assert(!JavascriptSymbol::Is(element));

PropertyDescriptor propertyDescriptor;

JavascriptConversion::ToPropertyKey(element, scriptContext, &propertyRecord);

if (JavascriptOperators::GetOwnPropertyDescriptor(RecyclableObject::FromVar(instance), propertyRecord->GetPropertyId(), scriptContext, &propertyDescriptor))

{

if (propertyDescriptor.IsEnumerable())

{

proxyResultToReturn->DirectSetItemAt(index++, element);

}

}

}

return proxyResultToReturn;

}

return JavascriptObject::CreateOwnEnumerableStringPropertiesHelper(object, scriptContext);

}

Var JavascriptOperators::GetOwnEnumerablePropertyNamesSymbols(Var instance, ScriptContext\* scriptContext)

{

RecyclableObject \*object = RecyclableObject::FromVar(ToObject(instance, scriptContext));

if (JavascriptProxy::Is(instance))

{

JavascriptProxy\* proxy = JavascriptProxy::FromVar(instance);

return proxy->PropertyKeysTrap(JavascriptProxy::KeysTrapKind::KeysKind);

}

return JavascriptObject::CreateOwnEnumerableStringSymbolPropertiesHelper(object, scriptContext);

}

BOOL JavascriptOperators::GetOwnProperty(Var instance, PropertyId propertyId, Var\* value, ScriptContext\* requestContext)

{

BOOL result;

if (TaggedNumber::Is(instance))

{

result = false;

}

else

{

RecyclableObject\* object = RecyclableObject::FromVar(instance);

result = object && object->GetProperty(object, propertyId, value, NULL, requestContext);

}

return result;

}

BOOL JavascriptOperators::GetOwnPropertyDescriptor(RecyclableObject\* obj, JavascriptString\* propertyKey, ScriptContext\* scriptContext, PropertyDescriptor\* propertyDescriptor)

{

return JavascriptOperators::GetOwnPropertyDescriptor(obj, JavascriptOperators::GetPropertyId(propertyKey, scriptContext), scriptContext, propertyDescriptor);

}

// ES5's [[GetOwnProperty]].

// Return value:

// FALSE means "undefined" PD.

// TRUE means success. The propertyDescriptor parameter gets the descriptor.

//

BOOL JavascriptOperators::GetOwnPropertyDescriptor(RecyclableObject\* obj, PropertyId propertyId, ScriptContext\* scriptContext, PropertyDescriptor\* propertyDescriptor)

{

Assert(obj);

Assert(scriptContext);

Assert(propertyDescriptor);

if (JavascriptProxy::Is(obj))

{

return JavascriptProxy::GetOwnPropertyDescriptor(obj, propertyId, scriptContext, propertyDescriptor);

}

Var getter, setter;

if (false == JavascriptOperators::GetOwnAccessors(obj, propertyId, &getter, &setter, scriptContext))

{

Var value;

if (false == JavascriptOperators::GetOwnProperty(obj, propertyId, &value, scriptContext))

{

return FALSE;

}

if (nullptr != value)

{

propertyDescriptor->SetValue(value);

}

//CONSIDER : Its expensive to query for each flag from type system. Combine this with the GetOwnProperty to get all the flags

//at once. This will require a new API from type system and override in all the types which overrides IsEnumerable etc.

//Currently there is no performance tuning for ES5. This should be ok.

propertyDescriptor->SetWritable(FALSE != obj->IsWritable(propertyId));

}

else

{

if (nullptr == getter)

{

getter = scriptContext->GetLibrary()->GetUndefined();

}

propertyDescriptor->SetGetter(getter);

if (nullptr == setter)

{

setter = scriptContext->GetLibrary()->GetUndefined();

}

propertyDescriptor->SetSetter(setter);

}

propertyDescriptor->SetConfigurable(FALSE != obj->IsConfigurable(propertyId));

propertyDescriptor->SetEnumerable(FALSE != obj->IsEnumerable(propertyId));

return TRUE;

}

\_\_inline RecyclableObject\* JavascriptOperators::GetPrototypeNoTrap(RecyclableObject\* instance)

{

Type\* type = instance->GetType();

if (type->HasSpecialPrototype())

{

if (type->GetTypeId() == TypeIds\_Proxy)

{

// get back null

Assert(type->GetPrototype() == instance->GetScriptContext()->GetLibrary()->GetNull());

return type->GetPrototype();

}

else

{

return instance->GetPrototypeSpecial();

}

}

return type->GetPrototype();

}

BOOL JavascriptOperators::IsArray(Var instanceVar)

{

if (!RecyclableObject::Is(instanceVar))

{

return FALSE;

}

RecyclableObject\* instance = RecyclableObject::FromVar(instanceVar);

if (DynamicObject::IsAnyArray(instance))

{

return TRUE;

}

if (JavascriptProxy::Is(instanceVar))

{

JavascriptProxy\* proxy = JavascriptProxy::FromVar(instanceVar);

return IsArray(proxy->GetTarget());

}

TypeId remoteTypeId = TypeIds\_Limit;

if (JavascriptOperators::GetRemoteTypeId(instanceVar, &remoteTypeId) &&

DynamicObject::IsAnyArrayTypeId(remoteTypeId))

{

return TRUE;

}

return FALSE;

}

BOOL JavascriptOperators::IsConstructor(Var instanceVar)

{

if (!RecyclableObject::Is(instanceVar))

{

return FALSE;

}

if (JavascriptProxy::Is(instanceVar))

{

JavascriptProxy\* proxy = JavascriptProxy::FromVar(instanceVar);

return IsConstructor(proxy->GetTarget());

}

if (!JavascriptFunction::Is(instanceVar))

{

return FALSE;

}

return JavascriptFunction::FromVar(instanceVar)->IsConstructor();

}

BOOL JavascriptOperators::IsConcatSpreadable(Var instanceVar)

{

// an object is spreadable under two condition, either it is a JsArray

// or you define an isconcatSpreadable flag on it.

if (!JavascriptOperators::IsObject(instanceVar))

{

return false;

}

RecyclableObject\* instance = RecyclableObject::FromVar(instanceVar);

ScriptContext\* scriptContext = instance->GetScriptContext();

Var spreadable = JavascriptOperators::GetProperty(instance, PropertyIds::\_symbolIsConcatSpreadable, scriptContext);

if (spreadable != scriptContext->GetLibrary()->GetUndefined())

{

return JavascriptConversion::ToBoolean(spreadable, scriptContext);

}

if (JavascriptOperators::IsArray(instance))

{

return true;

}

return false;

}

Var JavascriptOperators::OP\_LdCustomSpreadIteratorList(Var aRight, ScriptContext\* scriptContext)

{

RecyclableObject\* function = GetIteratorFunction(aRight, scriptContext);

JavascriptMethod method = function->GetEntryPoint();

if ((JavascriptArray::Is(aRight) && method == JavascriptArray::EntryInfo::Values.GetOriginalEntryPoint()) ||

(TypedArrayBase::Is(aRight) && method == TypedArrayBase::EntryInfo::Values.GetOriginalEntryPoint()))

{

// TODO: There is a compliance bug here in the case where the user has changed %ArrayIteratorPrototype%.next(); we won't call it.

// Checking if the property has been modified is currently not possible without doing a Get on it which might call user code.

// Fixing this bug will require a way to get the value stored in the property without doing the evaluation semantics of a Get.

return aRight;

}

Var iteratorVar = function->GetEntryPoint()(function, CallInfo(Js::CallFlags\_Value, 1), aRight);

if (!JavascriptOperators::IsObject(iteratorVar))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedObject);

}

RecyclableObject\* iterator = RecyclableObject::FromVar(iteratorVar);

return RecyclerNew(scriptContext->GetRecycler(), SpreadArgument, aRight, iterator, scriptContext->GetLibrary()->GetSpreadArgumentType());

}

BOOL JavascriptOperators::IsPropertyUnscopable(Var instanceVar, JavascriptString \*propertyString)

{

// This never gets called.

Throw::InternalError();

}

BOOL JavascriptOperators::IsPropertyUnscopable(Var instanceVar, PropertyId propertyId)

{

RecyclableObject\* instance = RecyclableObject::FromVar(instanceVar);

ScriptContext \* scriptContext = instance->GetScriptContext();

Var unscopables = JavascriptOperators::GetProperty(instance, PropertyIds::\_symbolUnscopables, scriptContext);

if (JavascriptOperators::IsObject(unscopables))

{

DynamicObject \*blackList = DynamicObject::FromVar(unscopables);

Var value;

//8.1.1.2.1.9.c If blocked is not undefined

if (JavascriptOperators::GetProperty(blackList, propertyId, &value, scriptContext))

{

return JavascriptConversion::ToBoolean(value, scriptContext);

}

}

return false;

}

BOOL JavascriptOperators::HasProperty(RecyclableObject\* instance, PropertyId propertyId)

{

while (JavascriptOperators::GetTypeId(instance) != TypeIds\_Null)

{

if (instance->HasProperty(propertyId))

{

return true;

}

instance = JavascriptOperators::GetPrototypeNoTrap(instance);

}

return false;

}

BOOL JavascriptOperators::HasPropertyUnscopables(RecyclableObject\* instance, PropertyId propertyId)

{

return JavascriptOperators::HasProperty(instance, propertyId)

&& !IsPropertyUnscopable(instance, propertyId);

}

BOOL JavascriptOperators::HasRootProperty(RecyclableObject\* instance, PropertyId propertyId)

{

Assert(RootObjectBase::Is(instance));

RootObjectBase\* rootObject = static\_cast<RootObjectBase\*>(instance);

if (rootObject->HasRootProperty(propertyId))

{

return true;

}

instance = instance->GetPrototype();

return HasProperty(instance, propertyId);

}

BOOL JavascriptOperators::HasProxyOrPrototypeInlineCacheProperty(RecyclableObject\* instance, PropertyId propertyId)

{

TypeId typeId;

typeId = JavascriptOperators::GetTypeId(instance);

if (typeId == Js::TypeIds\_Proxy)

{

// let's be more aggressive to disable inline prototype cache when proxy is presented in the prototypechain

return true;

}

do

{

instance = instance->GetPrototype();

typeId = JavascriptOperators::GetTypeId(instance);

if (typeId == Js::TypeIds\_Proxy)

{

// let's be more aggressive to disable inline prototype cache when proxy is presented in the prototypechain

return true;

}

if (typeId == TypeIds\_Null)

{

break;

}

/\* We can rule out object with deferred type handler, because they would have expanded if they are in the cache \*/

if (!instance->HasDeferredTypeHandler() && instance->HasProperty(propertyId)) { return true; }

} while (typeId != TypeIds\_Null);

return false;

}

BOOL JavascriptOperators::OP\_HasProperty(Var instance, PropertyId propertyId, ScriptContext\* scriptContext)

{

RecyclableObject\* object = TaggedNumber::Is(instance) ?

scriptContext->GetLibrary()->GetNumberPrototype() :

RecyclableObject::FromVar(instance);

BOOL result = HasProperty(object, propertyId);

return result;

}

BOOL JavascriptOperators::OP\_HasOwnProperty(Var instance, PropertyId propertyId, ScriptContext\* scriptContext)

{

RecyclableObject\* object = TaggedNumber::Is(instance) ?

scriptContext->GetLibrary()->GetNumberPrototype() :

RecyclableObject::FromVar(instance);

BOOL result = HasOwnProperty(object, propertyId, scriptContext);

return result;

}

// CONSIDER: Have logic similar to HasOwnPropertyNoHostObjectForHeapEnum

BOOL JavascriptOperators::HasOwnPropertyNoHostObject(Var instance, PropertyId propertyId)

{

AssertMsg(!TaggedNumber::Is(instance), "HasOwnPropertyNoHostObject int passed");

RecyclableObject\* object = RecyclableObject::FromVar(instance);

return object && object->HasOwnPropertyNoHostObject(propertyId);

}

// CONSIDER: Remove HasOwnPropertyNoHostObjectForHeapEnum and use GetOwnPropertyNoHostObjectForHeapEnum in its place by changing it

// to return BOOL, true or false with whether the property exists or not, and return the value if not getter/setter as an out param.

BOOL JavascriptOperators::HasOwnPropertyNoHostObjectForHeapEnum(Var instance, PropertyId propertyId, ScriptContext\* requestContext, Var& getter, Var& setter)

{

AssertMsg(!TaggedNumber::Is(instance), "HasOwnPropertyNoHostObjectForHeapEnum int passed");

RecyclableObject \* object = RecyclableObject::FromVar(instance);

if (StaticType::Is(object->GetTypeId()))

{

return FALSE;

}

getter = setter = NULL;

DynamicObject\* dynamicObject = DynamicObject::FromVar(instance);

Assert(dynamicObject->GetScriptContext()->IsHeapEnumInProgress());

if (dynamicObject->UseDynamicObjectForNoHostObjectAccess())

{

if (!dynamicObject->DynamicObject::GetAccessors(propertyId, &getter, &setter, requestContext))

{

Var value;

if (!dynamicObject->DynamicObject::GetProperty(instance, propertyId, &value, NULL, requestContext) ||

(requestContext->IsUndeclBlockVar(value) && (ActivationObject::Is(instance) || RootObjectBase::Is(instance))))

{

return FALSE;

}

}

}

else

{

if (!object->GetAccessors(propertyId, &getter, &setter, requestContext))

{

Var value;

if (!object->GetProperty(instance, propertyId, &value, NULL, requestContext) ||

(requestContext->IsUndeclBlockVar(value) && (ActivationObject::Is(instance) || RootObjectBase::Is(instance))))

{

return FALSE;

}

}

}

return TRUE;

}

Var JavascriptOperators::GetOwnPropertyNoHostObjectForHeapEnum(Var instance, PropertyId propertyId, ScriptContext\* requestContext, Var& getter, Var& setter)

{

AssertMsg(!TaggedNumber::Is(instance), "GetDataPropertyNoHostObject int passed");

Assert(HasOwnPropertyNoHostObjectForHeapEnum(instance, propertyId, requestContext, getter, setter) || getter || setter);

DynamicObject\* dynamicObject = DynamicObject::FromVar(instance);

getter = setter = NULL;

if (NULL == dynamicObject)

{

return requestContext->GetLibrary()->GetUndefined();

}

Var returnVar = requestContext->GetLibrary()->GetUndefined();

BOOL result = FALSE;

if (dynamicObject->UseDynamicObjectForNoHostObjectAccess())

{

if (! dynamicObject->DynamicObject::GetAccessors(propertyId, &getter, &setter, requestContext))

{

result = dynamicObject->DynamicObject::GetProperty(instance, propertyId, &returnVar, NULL, requestContext);

}

}

else

{

if (! dynamicObject->GetAccessors(propertyId, &getter, &setter, requestContext))

{

result = dynamicObject->GetProperty(instance, propertyId, &returnVar, NULL, requestContext);

}

}

if (result)

{

return returnVar;

}

return requestContext->GetLibrary()->GetUndefined();

}

BOOL JavascriptOperators::OP\_HasOwnPropScoped(Var scope, PropertyId propertyId, Var defaultInstance, ScriptContext\* scriptContext)

{

AssertMsg(scope == scriptContext->GetLibrary()->GetNull() || JavascriptArray::Is(scope),

"Invalid scope chain pointer passed - should be null or an array");

if (JavascriptArray::Is(scope))

{

JavascriptArray\* arrScope = JavascriptArray::FromVar(scope);

Var instance = arrScope->DirectGetItem(0);

return JavascriptOperators::OP\_HasOwnProperty(instance, propertyId, scriptContext);

}

return JavascriptOperators::OP\_HasOwnProperty(defaultInstance, propertyId, scriptContext);

}

BOOL JavascriptOperators::GetPropertyUnscopable(Var instance, RecyclableObject\* propertyObject, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

return GetProperty\_Internal<true>(instance, propertyObject, false, propertyId, value, requestContext, info);

}

BOOL JavascriptOperators::GetProperty(Var instance, RecyclableObject\* propertyObject, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

return GetProperty\_Internal<false>(instance, propertyObject, false, propertyId, value, requestContext, info);

}

BOOL JavascriptOperators::GetRootProperty(Var instance, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

return GetProperty\_Internal<false>(instance, RecyclableObject::FromVar(instance), true, propertyId, value, requestContext, info);

}

template <bool unscopables>

BOOL JavascriptOperators::GetProperty\_Internal(Var instance, RecyclableObject\* propertyObject, const bool isRoot, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

if (TaggedNumber::Is(instance))

{

PropertyValueInfo::ClearCacheInfo(info);

}

RecyclableObject\* object = propertyObject;

BOOL foundProperty = FALSE;

if (isRoot)

{

Assert(RootObjectBase::Is(object));

RootObjectBase\* rootObject = static\_cast<RootObjectBase\*>(object);

foundProperty = rootObject->GetRootProperty(instance, propertyId, value, info, requestContext);

}

while (!foundProperty && JavascriptOperators::GetTypeId(object) != TypeIds\_Null)

{

if (unscopables && IsPropertyUnscopable(object, propertyId))

{

break;

}

else

{

if (object->GetProperty(instance, propertyId, value, info, requestContext))

{

foundProperty = true;

break;

}

}

if (object->SkipsPrototype())

{

break;

}

object = JavascriptOperators::GetPrototypeNoTrap(object);

}

if (foundProperty)

{

#if DBG

if (DynamicObject::Is(object))

{

DynamicObject\* dynamicObject = (DynamicObject\*)object;

DynamicTypeHandler\* dynamicTypeHandler = dynamicObject->GetDynamicType()->GetTypeHandler();

Var property;

if (dynamicTypeHandler->CheckFixedProperty(requestContext->GetPropertyName(propertyId), &property, requestContext))

{

Assert(value == nullptr || \*value == property);

}

}

#endif

// Don't cache the information if the value is undecl block var

// REVIEW: We might want to only check this if we need to (For LdRootFld or ScopedLdFld)

// Also we might want to throw here instead of checking it again in the caller

if (value && !requestContext->IsUndeclBlockVar(\*value) && !WithScopeObject::Is(object))

{

CacheOperators::CachePropertyRead(instance, object, isRoot, propertyId, false, info, requestContext);

}

#ifdef TELEMETRY\_JSO

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

requestContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(instance, propertyId, value, /\*successful: \*/true);

}

#endif

return TRUE;

}

else

{

#ifdef MISSING\_PROPERTY\_STATS

if (PHASE\_STATS1(MissingPropertyCachePhase))

{

requestContext->RecordMissingPropertyMiss();

}

#endif

if (PHASE\_TRACE1(MissingPropertyCachePhase))

{

Output::Print(L"MissingPropertyCaching: Missing property %d on slow path.\n", propertyId);

}

// Only cache missing property lookups for non-root field loads on objects that have PathTypeHandlers, because only these objects guarantee a type change when the property is added,

// which obviates the need to explicitly invalidate missing property inline caches.

if (!PHASE\_OFF1(MissingPropertyCachePhase) && !isRoot && DynamicObject::Is(instance) && ((DynamicObject\*)instance)->GetDynamicType()->GetTypeHandler()->IsPathTypeHandler())

{

#ifdef MISSING\_PROPERTY\_STATS

if (PHASE\_STATS1(MissingPropertyCachePhase))

{

requestContext->RecordMissingPropertyCacheAttempt();

}

#endif

if (PHASE\_TRACE1(MissingPropertyCachePhase))

{

Output::Print(L"MissingPropertyCache: Caching missing property for property %d.\n", propertyId);

}

PropertyValueInfo::Set(info, requestContext->GetLibrary()->GetMissingPropertyHolder(), 0);

CacheOperators::CachePropertyRead(instance, requestContext->GetLibrary()->GetMissingPropertyHolder(), isRoot, propertyId, true, info, requestContext);

}

#if defined(TELEMETRY\_JSO) || defined(TELEMETRY\_AddToCache) // enabled for `TELEMETRY\_AddToCache`, because this is the property-not-found codepath where the normal TELEMETRY\_AddToCache code wouldn't be executed.

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

if (info && info->AllowResizingPolymorphicInlineCache()) // If in interpreted mode, not JIT.

{

requestContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(instance, propertyId, nullptr, /\*successful: \*/false);

}

}

#endif

return FALSE;

}

}

template<typename PropertyKeyType>

BOOL JavascriptOperators::GetPropertyWPCache(Var instance, RecyclableObject\* propertyObject, PropertyKeyType propertyKey, Var\* value, ScriptContext\* requestContext, PropertyString \* propertyString)

{

if (TaggedNumber::Is(instance))

{

propertyString = NULL;

}

PropertyValueInfo info;

RecyclableObject\* object = propertyObject;

while (JavascriptOperators::GetTypeId(object) != TypeIds\_Null)

{

if (object->GetProperty(instance, propertyKey, value, &info, requestContext))

{

if (propertyString != NULL)

{

uint16 slotIndex = info.GetPropertyIndex();

if (slotIndex != Constants::NoSlot &&

info.GetInstance() == object &&

info.IsWritable() && !object->CanHaveInterceptors() &&

requestContext == object->GetScriptContext() &&

((info.GetFlags() & (InlineCacheGetterFlag | InlineCacheSetterFlag)) == 0))

{

uint16 inlineOrAuxSlotIndex;

bool isInlineSlot;

DynamicObject::FromVar(info.GetInstance())->GetTypeHandler()->PropertyIndexToInlineOrAuxSlotIndex(slotIndex, &inlineOrAuxSlotIndex, &isInlineSlot);

propertyString->UpdateCache(info.GetInstance()->GetType(), inlineOrAuxSlotIndex, isInlineSlot, info.IsStoreFieldCacheEnabled());

}

}

return TRUE;

}

if (object->SkipsPrototype())

{

break;

}

object = JavascriptOperators::GetPrototypeNoTrap(object);

}

return FALSE;

}

BOOL JavascriptOperators::GetPropertyObject(Var instance, ScriptContext \* scriptContext, RecyclableObject\*\* propertyObject)

{

Assert(propertyObject);

if (TaggedNumber::Is(instance))

{

\*propertyObject = scriptContext->GetLibrary()->GetNumberPrototype();

return TRUE;

}

RecyclableObject\* object = RecyclableObject::FromVar(instance);

TypeId typeId = object->GetTypeId();

\*propertyObject = object;

if (typeId == TypeIds\_Null || typeId == TypeIds\_Undefined)

{

return FALSE;

}

return TRUE;

}

#if DBG

BOOL JavascriptOperators::IsPropertyObject(RecyclableObject \* instance)

{

TypeId typeId = JavascriptOperators::GetTypeId(instance);

return (typeId != TypeIds\_Integer && typeId != TypeIds\_Null && typeId != TypeIds\_Undefined);

}

#endif

Var JavascriptOperators::OP\_GetProperty(Var instance, PropertyId propertyId, ScriptContext\* scriptContext)

{

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined, scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

else

{

return scriptContext->GetLibrary()->GetUndefined();

}

}

Var result = JavascriptOperators::GetProperty(instance, object, propertyId, scriptContext);

AssertMsg(result != nullptr, "result null in OP\_GetProperty");

return result;

}

Var JavascriptOperators::OP\_GetRootProperty(Var instance, PropertyId propertyId, PropertyValueInfo \* info, ScriptContext\* scriptContext)

{

AssertMsg(RootObjectBase::Is(instance), "Root must be an object!");

Var value;

if (JavascriptOperators::GetRootProperty(RecyclableObject::FromVar(instance), propertyId, &value, scriptContext, info))

{

if (scriptContext->IsUndeclBlockVar(value))

{

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UseBeforeDeclaration);

}

return value;

}

const wchar\_t\* propertyName = scriptContext->GetPropertyName(propertyId)->GetBuffer();

JavascriptFunction \* caller = nullptr;

if (JavascriptStackWalker::GetCaller(&caller, scriptContext))

{

FunctionBody \* callerBody = caller->GetFunctionBody();

if (callerBody && callerBody->GetUtf8SourceInfo()->GetIsXDomain())

{

propertyName = nullptr;

}

}

// Don't error if we disabled implicit calls

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UndefVariable, propertyName);

}

return scriptContext->GetLibrary()->GetUndefined();

}

Var JavascriptOperators::OP\_GetThisScoped(FrameDisplay \*pScope, Var defaultInstance, ScriptContext\* scriptContext)

{

// NOTE: If changes are made to this logic be sure to update the debuggers as well

int length = pScope->GetLength();

for (int i = 0; i < length; i += 1)

{

Var value;

DynamicObject \*obj = DynamicObject::FromVar(pScope->GetItem(i));

if (JavascriptOperators::GetProperty(obj, Js::PropertyIds::\_lexicalThisSlotSymbol, &value, scriptContext))

{

return value;

}

}

return defaultInstance;

}

Var JavascriptOperators::OP\_UnwrapWithObj(Var aValue)

{

return RecyclableObject::FromVar(aValue)->GetThisObjectOrUnWrap();

}

Var JavascriptOperators::OP\_GetInstanceScoped(FrameDisplay \*pScope, PropertyId propertyId, Var rootObject, Var\* thisVar, ScriptContext\* scriptContext)

{

// Similar to GetPropertyScoped, but instead of returning the property value, we return the instance that

// owns it, or the global object if no instance is found.

int i;

int length = pScope->GetLength();

for (i = 0; i < length; i++)

{

RecyclableObject \*obj = (RecyclableObject\*)pScope->GetItem(i);

if (JavascriptOperators::HasProperty(obj, propertyId))

{

// HasProperty will call WithObjects HasProperty which will do the filtering

// All we have to do here is unwrap the object hence the api call

\*thisVar = obj->GetThisObjectOrUnWrap();

return \*thisVar;

}

}

\*thisVar = scriptContext->GetLibrary()->GetUndefined();

if (rootObject != scriptContext->GetGlobalObject())

{

if (JavascriptOperators::OP\_HasProperty(rootObject, propertyId, scriptContext))

{

return rootObject;

}

}

return scriptContext->GetGlobalObject();

}

Var JavascriptOperators::GetPropertyReference(RecyclableObject \*instance, PropertyId propertyId, ScriptContext\* requestContext)

{

Var value = nullptr;

PropertyValueInfo info;

if (JavascriptOperators::GetPropertyReference(instance, propertyId, &value, requestContext, &info))

{

Assert(value != nullptr);

return value;

}

return requestContext->GetLibrary()->GetUndefined();

}

BOOL JavascriptOperators::GetPropertyReference(Var instance, RecyclableObject\* propertyObject, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

return GetPropertyReference\_Internal(instance, propertyObject, false, propertyId, value, requestContext, info);

}

BOOL JavascriptOperators::GetRootPropertyReference(RecyclableObject\* instance, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

return GetPropertyReference\_Internal(instance, instance, true, propertyId, value, requestContext, info);

}

BOOL JavascriptOperators::PropertyReferenceWalkUnscopable(Var instance, RecyclableObject\*\* propertyObject, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

return PropertyReferenceWalk\_Impl<true>(instance, propertyObject, propertyId, value, info, requestContext);

}

BOOL JavascriptOperators::PropertyReferenceWalk(Var instance, RecyclableObject\*\* propertyObject, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

return PropertyReferenceWalk\_Impl<false>(instance, propertyObject, propertyId, value, info, requestContext);

}

template <bool unscopables>

BOOL JavascriptOperators::PropertyReferenceWalk\_Impl(Var instance, RecyclableObject\*\* propertyObject, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

BOOL foundProperty = false;

RecyclableObject\* object = \*propertyObject;

while (!foundProperty && JavascriptOperators::GetTypeId(object) != TypeIds\_Null)

{

if (unscopables && JavascriptOperators::IsPropertyUnscopable(object, propertyId))

{

break;

}

else

{

if (object->GetPropertyReference(instance, propertyId, value, info, requestContext))

{

foundProperty = true;

break;

}

}

if (object->SkipsPrototype())

{

break; // will return false

}

object = JavascriptOperators::GetPrototypeNoTrap(object);

}

\*propertyObject = object;

return foundProperty;

}

BOOL JavascriptOperators::GetPropertyReference\_Internal(Var instance, RecyclableObject\* propertyObject, const bool isRoot, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

if (TaggedNumber::Is(instance))

{

PropertyValueInfo::ClearCacheInfo(info);

}

BOOL foundProperty = FALSE;

RecyclableObject\* object = propertyObject;

if (isRoot)

{

foundProperty = RootObjectBase::FromVar(object)->GetRootPropertyReference(instance, propertyId, value, info, requestContext);

}

if (!foundProperty)

{

foundProperty = PropertyReferenceWalk(instance, &object, propertyId, value, info, requestContext);

}

if (!foundProperty)

{

#if defined(TELEMETRY\_JSO) || defined(TELEMETRY\_AddToCache) // enabled for `TELEMETRY\_AddToCache`, because this is the property-not-found codepath where the normal TELEMETRY\_AddToCache code wouldn't be executed.

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

if (info && info->AllowResizingPolymorphicInlineCache()) // If in interpreted mode, not JIT.

{

requestContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(instance, propertyId, nullptr, /\*successful: \*/false);

}

}

#endif

return foundProperty;

}

if (requestContext->IsUndeclBlockVar(\*value))

{

JavascriptError::ThrowReferenceError(requestContext, JSERR\_UseBeforeDeclaration);

}

#if DBG

if (DynamicObject::Is(object))

{

DynamicObject\* dynamicObject = (DynamicObject\*)object;

DynamicTypeHandler\* dynamicTypeHandler = dynamicObject->GetDynamicType()->GetTypeHandler();

Var property;

if (dynamicTypeHandler->CheckFixedProperty(requestContext->GetPropertyName(propertyId), &property, requestContext))

{

Assert(value == nullptr || \*value == property);

}

}

#endif

CacheOperators::CachePropertyRead(instance, object, isRoot, propertyId, false, info, requestContext);

return TRUE;

}

template <typename PropertyKeyType, bool unscopable>

DescriptorFlags JavascriptOperators::GetterSetter\_Impl(RecyclableObject\* instance, PropertyKeyType propertyKey, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* scriptContext)

{

DescriptorFlags flags = None;

RecyclableObject\* object = instance;

while (flags == None && JavascriptOperators::GetTypeId(object) != TypeIds\_Null)

{

if (unscopable && IsPropertyUnscopable(object, propertyKey))

{

break;

}

else

{

flags = object->GetSetter(propertyKey, setterValue, info, scriptContext);

if (flags != None)

{

break;

}

}

// CONSIDER: we should add SkipsPrototype support. DOM has no ES 5 concepts built in that aren't

// already part of our prototype objects which are chakra objects.

object = object->GetPrototype();

}

return flags;

}

DescriptorFlags JavascriptOperators::GetterSetterUnscopable(RecyclableObject\* instance, PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* scriptContext)

{

return GetterSetter\_Impl<PropertyId, true>(instance, propertyId, setterValue, info, scriptContext);

}

DescriptorFlags JavascriptOperators::GetterSetter(RecyclableObject\* instance, PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* scriptContext)

{

return GetterSetter\_Impl<PropertyId, false>(instance, propertyId, setterValue, info, scriptContext);

}

DescriptorFlags JavascriptOperators::GetterSetter(RecyclableObject\* instance, JavascriptString \* propertyName, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* scriptContext)

{

return GetterSetter\_Impl<JavascriptString\*, false>(instance, propertyName, setterValue, info, scriptContext);

}

// Checks to see if any object in the prototype chain has a property descriptor for the given property

// that specifies either an accessor or a non-writable attribute.

// If TRUE, check flags for details.

template<typename PropertyKeyType, bool doFastProtoChainCheck, bool isRoot>

BOOL JavascriptOperators::CheckPrototypesForAccessorOrNonWritablePropertyCore(RecyclableObject\* instance,

PropertyKeyType propertyKey, Var\* setterValue, DescriptorFlags\* flags, PropertyValueInfo\* info, ScriptContext\* scriptContext)

{

Assert(setterValue);

Assert(flags);

// Do a quick check to see if all objects in the prototype chain are known to have only

// writable data properties (i.e. no accessors or non-writable properties).

if (doFastProtoChainCheck && CheckIfObjectAndPrototypeChainHasOnlyWritableDataProperties(instance))

{

return FALSE;

}

if (isRoot)

{

\*flags = JavascriptOperators::GetRootSetter(instance, propertyKey, setterValue, info, scriptContext);

}

if (\*flags == None)

{

\*flags = JavascriptOperators::GetterSetter(instance, propertyKey, setterValue, info, scriptContext);

}

return ((\*flags & Accessor) == Accessor) || ((\*flags & Proxy) == Proxy)|| ((\*flags & Data) == Data && (\*flags & Writable) == None);

}

void JavascriptOperators::OP\_InvalidateProtoCaches(PropertyId propertyId, ScriptContext \*scriptContext)

{

scriptContext->InvalidateProtoCaches(propertyId);

}

// Checks to see if any object in the prototype chain has a property descriptor for the given index

// that specifies either an accessor or a non-writable attribute.

// If TRUE, check flags for details.

BOOL JavascriptOperators::CheckPrototypesForAccessorOrNonWritableItem(RecyclableObject\* instance, uint32 index,

Var\* setterValue, DescriptorFlags \*flags, ScriptContext\* scriptContext, BOOL skipPrototypeCheck /\* = FALSE \*/)

{

Assert(setterValue);

Assert(flags);

// Do a quick walk up the prototype chain to see if any of the prototypes has ever had ANY setter or non-writable property.

if (CheckIfObjectAndPrototypeChainHasOnlyWritableDataProperties(instance))

{

return FALSE;

}

RecyclableObject\* object = instance;

while (JavascriptOperators::GetTypeId(object) != TypeIds\_Null)

{

\*flags = object->GetItemSetter(index, setterValue, scriptContext);

if (\*flags != None || skipPrototypeCheck)

{

break;

}

object = object->GetPrototype();

}

return ((\*flags & Accessor) == Accessor) || ((\*flags & Proxy) == Proxy) || ((\*flags & Data) == Data && (\*flags & Writable) == None);

}

BOOL JavascriptOperators::SetGlobalPropertyNoHost(wchar\_t const \* propertyName, charcount\_t propertyLength, Var value, ScriptContext \* scriptContext)

{

GlobalObject \* globalObject = scriptContext->GetGlobalObject();

uint32 index;

PropertyRecord const \* propertyRecord;

IndexType indexType = GetIndexTypeFromString(propertyName, propertyLength, scriptContext, &index, &propertyRecord, true);

if (indexType == IndexType\_Number)

{

return globalObject->DynamicObject::SetItem(index, value, PropertyOperation\_None);

}

return globalObject->DynamicObject::SetProperty(propertyRecord->GetPropertyId(), value, PropertyOperation\_None, NULL);

}

template<typename PropertyKeyType>

BOOL JavascriptOperators::SetPropertyWPCache(Var receiver, RecyclableObject\* object, PropertyKeyType propertyKey, Var newValue, ScriptContext\* requestContext, PropertyString \* propertyString, PropertyOperationFlags propertyOperationFlags)

{

if (receiver)

{

AnalysisAssert(object);

Assert(!TaggedNumber::Is(receiver));

Var setterValueOrProxy = nullptr;

DescriptorFlags flags = None;

if (JavascriptOperators::CheckPrototypesForAccessorOrNonWritableProperty(object, propertyKey, &setterValueOrProxy, &flags, NULL, requestContext))

{

if ((flags & Accessor) == Accessor)

{

if (JavascriptError::ThrowIfStrictModeUndefinedSetter(propertyOperationFlags, setterValueOrProxy, requestContext))

{

return TRUE;

}

if (setterValueOrProxy)

{

receiver = (RecyclableObject::FromVar(receiver))->GetThisObjectOrUnWrap();

RecyclableObject\* func = RecyclableObject::FromVar(setterValueOrProxy);

JavascriptOperators::CallSetter(func, receiver, newValue, requestContext);

}

return TRUE;

}

else if ((flags & Proxy) == Proxy)

{

Assert(JavascriptProxy::Is(setterValueOrProxy));

JavascriptProxy\* proxy = JavascriptProxy::FromVar(setterValueOrProxy);

auto fn = [&](RecyclableObject\* target) -> BOOL {

return JavascriptOperators::SetPropertyWPCache(receiver, target, propertyKey, newValue, requestContext, propertyString, propertyOperationFlags);

};

return proxy->SetPropertyTrap(receiver, JavascriptProxy::SetPropertyTrapKind::SetPropertyWPCacheKind, propertyKey, newValue, requestContext);

}

else

{

Assert((flags & Data) == Data && (flags & Writable) == None);

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, requestContext);

requestContext->GetThreadContext()->AddImplicitCallFlags(ImplicitCall\_NoOpSet);

return FALSE;

}

}

else if (!JavascriptOperators::IsObject(receiver))

{

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, requestContext);

return FALSE;

}

RecyclableObject\* receiverObject = RecyclableObject::FromVar(receiver);

if (receiver != object)

{

// If the receiver object has the property and it is an accessor then return false

PropertyDescriptor existingDesc;

if (JavascriptOperators::GetOwnPropertyDescriptor(receiverObject, propertyKey, requestContext, &existingDesc)

&& existingDesc.IsAccessorDescriptor())

{

return FALSE;

}

}

// in 9.1.9, step 5, we should return false if receiver is not object, and that will happen in default RecyclableObject operation anyhow.

PropertyValueInfo info;

if (receiverObject->SetProperty(propertyKey, newValue, propertyOperationFlags, &info))

{

if (propertyString != NULL)

{

uint16 slotIndex = info.GetPropertyIndex();

if (slotIndex != Constants::NoSlot &&

info.GetInstance() == receiverObject &&

!object->CanHaveInterceptors() &&

requestContext == receiverObject->GetScriptContext() &&

(info.GetFlags() != InlineCacheSetterFlag))

{

uint16 inlineOrAuxSlotIndex;

bool isInlineSlot;

DynamicObject::FromVar(info.GetInstance())->GetTypeHandler()->PropertyIndexToInlineOrAuxSlotIndex(info.GetPropertyIndex(), &inlineOrAuxSlotIndex, &isInlineSlot);

propertyString->UpdateCache(info.GetInstance()->GetType(), inlineOrAuxSlotIndex, isInlineSlot, info.IsStoreFieldCacheEnabled());

}

}

return TRUE;

}

}

return FALSE;

}

BOOL JavascriptOperators::SetItemOnTaggedNumber(Var receiver, RecyclableObject\* object, uint32 index, Var newValue, ScriptContext\* requestContext,

PropertyOperationFlags propertyOperationFlags)

{

Assert(TaggedNumber::Is(receiver));

if (requestContext->optimizationOverrides.GetSideEffects() & SideEffects\_Accessor)

{

Var setterValueOrProxy = nullptr;

DescriptorFlags flags = None;

if (object == nullptr)

{

GetPropertyObject(receiver, requestContext, &object);

}

if (JavascriptOperators::CheckPrototypesForAccessorOrNonWritableItem(object, index, &setterValueOrProxy, &flags, requestContext))

{

if ((flags & Accessor) == Accessor)

{

if (JavascriptError::ThrowIfStrictModeUndefinedSetter(propertyOperationFlags, setterValueOrProxy, requestContext))

{

return TRUE;

}

if (setterValueOrProxy)

{

RecyclableObject\* func = RecyclableObject::FromVar(setterValueOrProxy);

JavascriptOperators::CallSetter(func, receiver, newValue, requestContext);

return TRUE;

}

}

else if ((flags & Proxy) == Proxy)

{

Assert(JavascriptProxy::Is(setterValueOrProxy));

JavascriptProxy\* proxy = JavascriptProxy::FromVar(setterValueOrProxy);

const PropertyRecord\* propertyRecord;

proxy->PropertyIdFromInt(index, &propertyRecord);

return proxy->SetPropertyTrap(receiver, JavascriptProxy::SetPropertyTrapKind::SetItemOnTaggedNumberKind, propertyRecord->GetPropertyId(), newValue, requestContext);

}

else

{

Assert((flags & Data) == Data && (flags & Writable) == None);

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, requestContext);

}

}

}

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, requestContext);

return FALSE;

}

BOOL JavascriptOperators::SetPropertyOnTaggedNumber(Var receiver, RecyclableObject\* object, PropertyId propertyId, Var newValue, ScriptContext\* requestContext,

PropertyOperationFlags propertyOperationFlags)

{

Assert (TaggedNumber::Is(receiver));

if (requestContext->optimizationOverrides.GetSideEffects() & SideEffects\_Accessor)

{

Var setterValueOrProxy = nullptr;

PropertyValueInfo info;

DescriptorFlags flags = None;

if (object == nullptr)

{

GetPropertyObject(receiver, requestContext, &object);

}

if (JavascriptOperators::CheckPrototypesForAccessorOrNonWritableProperty(object, propertyId, &setterValueOrProxy, &flags, &info, requestContext))

{

if ((flags & Accessor) == Accessor)

{

if (JavascriptError::ThrowIfStrictModeUndefinedSetter(propertyOperationFlags, setterValueOrProxy, requestContext))

{

return TRUE;

}

if (setterValueOrProxy)

{

RecyclableObject\* func = RecyclableObject::FromVar(setterValueOrProxy);

Assert(info.GetFlags() == InlineCacheSetterFlag || info.GetPropertyIndex() == Constants::NoSlot);

JavascriptOperators::CallSetter(func, receiver, newValue, requestContext);

return TRUE;

}

}

else if ((flags & Proxy) == Proxy)

{

Assert(JavascriptProxy::Is(setterValueOrProxy));

JavascriptProxy\* proxy = JavascriptProxy::FromVar(setterValueOrProxy);

return proxy->SetPropertyTrap(receiver, JavascriptProxy::SetPropertyTrapKind::SetPropertyOnTaggedNumberKind, propertyId, newValue, requestContext);

}

else

{

Assert((flags & Data) == Data && (flags & Writable) == None);

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, requestContext);

}

}

}

// Add implicit call flags, to bail out if field copy prop may propagate the wrong value.

requestContext->GetThreadContext()->AddImplicitCallFlags(ImplicitCall\_NoOpSet);

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, requestContext);

return FALSE;

}

BOOL JavascriptOperators::SetPropertyUnscopable(Var instance, RecyclableObject\* receiver, PropertyId propertyId, Var newValue, PropertyValueInfo \* info, ScriptContext\* requestContext, PropertyOperationFlags propertyOperationFlags)

{

return SetProperty\_Internal<true>(instance, receiver, false, propertyId, newValue, info, requestContext, propertyOperationFlags);

}

BOOL JavascriptOperators::SetProperty(Var receiver, RecyclableObject\* object, PropertyId propertyId, Var newValue, PropertyValueInfo \* info, ScriptContext\* requestContext, PropertyOperationFlags propertyOperationFlags)

{

return SetProperty\_Internal<false>(receiver, object, false, propertyId, newValue, info, requestContext, propertyOperationFlags);

}

BOOL JavascriptOperators::SetRootProperty(RecyclableObject\* instance, PropertyId propertyId, Var newValue, PropertyValueInfo \* info, ScriptContext\* requestContext, PropertyOperationFlags propertyOperationFlags)

{

return SetProperty\_Internal<false>(instance, instance, true, propertyId, newValue, info, requestContext, propertyOperationFlags);

}

template <bool unscopables>

BOOL JavascriptOperators::SetProperty\_Internal(Var receiver, RecyclableObject\* object, const bool isRoot, PropertyId propertyId, Var newValue, PropertyValueInfo \* info, ScriptContext\* requestContext, PropertyOperationFlags propertyOperationFlags)

{

if (receiver)

{

Assert(!TaggedNumber::Is(receiver));

Var setterValueOrProxy = nullptr;

DescriptorFlags flags = None;

if ((isRoot && JavascriptOperators::CheckPrototypesForAccessorOrNonWritableRootProperty(object, propertyId, &setterValueOrProxy, &flags, info, requestContext)) ||

(!isRoot && JavascriptOperators::CheckPrototypesForAccessorOrNonWritableProperty(object, propertyId, &setterValueOrProxy, &flags, info, requestContext)))

{

if ((flags & Accessor) == Accessor)

{

if (JavascriptError::ThrowIfStrictModeUndefinedSetter(propertyOperationFlags, setterValueOrProxy, requestContext) ||

JavascriptError::ThrowIfNotExtensibleUndefinedSetter(propertyOperationFlags, setterValueOrProxy, requestContext))

{

return TRUE;

}

if (setterValueOrProxy)

{

RecyclableObject\* func = RecyclableObject::FromVar(setterValueOrProxy);

Assert(!info || info->GetFlags() == InlineCacheSetterFlag || info->GetPropertyIndex() == Constants::NoSlot);

if (WithScopeObject::Is(receiver))

{

receiver = (RecyclableObject::FromVar(receiver))->GetThisObjectOrUnWrap();

}

else

{

CacheOperators::CachePropertyWrite(RecyclableObject::FromVar(receiver), isRoot, object->GetType(), propertyId, info, requestContext);

}

#ifdef ENABLE\_MUTATION\_BREAKPOINT

if (MutationBreakpoint::IsFeatureEnabled(requestContext))

{

MutationBreakpoint::HandleSetProperty(requestContext, object, propertyId, newValue);

}

#endif

JavascriptOperators::CallSetter(func, receiver, newValue, requestContext);

}

return TRUE;

}

else if ((flags & Proxy) == Proxy)

{

Assert(JavascriptProxy::Is(setterValueOrProxy));

JavascriptProxy\* proxy = JavascriptProxy::FromVar(setterValueOrProxy);

// We can't cache the property at this time. both target and handler can be changed outside of the proxy, so the inline cache needs to be

// invalidate when target, handler, or handler prototype has changed. We don't have a way to achieve this yet.

PropertyValueInfo::SetNoCache(info, proxy);

PropertyValueInfo::DisablePrototypeCache(info, proxy); // We can't cache prototype property either

return proxy->SetPropertyTrap(receiver, JavascriptProxy::SetPropertyTrapKind::SetPropertyKind, propertyId, newValue, requestContext);

}

else

{

Assert((flags & Data) == Data && (flags & Writable) == None);

if (flags & Const)

{

JavascriptError::ThrowReferenceError(requestContext, ERRAssignmentToConst);

}

JavascriptError::ThrowCantAssign(propertyOperationFlags, requestContext, propertyId);

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, requestContext);

return FALSE;

}

}

else if (!JavascriptOperators::IsObject(receiver))

{

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, requestContext);

return FALSE;

}

#ifdef ENABLE\_MUTATION\_BREAKPOINT

// Break on mutation if needed

bool doNotUpdateCacheForMbp = MutationBreakpoint::IsFeatureEnabled(requestContext) ?

MutationBreakpoint::HandleSetProperty(requestContext, object, propertyId, newValue) : false;

#endif

// Get the original type before setting the property

Type \*typeWithoutProperty = object->GetType();

BOOL didSetProperty = false;

if (isRoot)

{

AssertMsg(JavascriptOperators::GetTypeId(receiver) == TypeIds\_GlobalObject

|| JavascriptOperators::GetTypeId(receiver) == TypeIds\_ModuleRoot,

"Root must be a global object!");

RootObjectBase\* rootObject = static\_cast<RootObjectBase\*>(receiver);

didSetProperty = rootObject->SetRootProperty(propertyId, newValue, propertyOperationFlags, info);

}

else

{

RecyclableObject\* instanceObject = RecyclableObject::FromVar(receiver);

while (JavascriptOperators::GetTypeId(instanceObject) != TypeIds\_Null)

{

if (unscopables && JavascriptOperators::IsPropertyUnscopable(instanceObject, propertyId))

{

break;

}

else

{

didSetProperty = instanceObject->SetProperty(propertyId, newValue, propertyOperationFlags, info);

if (didSetProperty || !unscopables)

{

break;

}

}

instanceObject = JavascriptOperators::GetPrototypeNoTrap(instanceObject);

}

}

if (didSetProperty)

{

bool updateCache = true;

#ifdef ENABLE\_MUTATION\_BREAKPOINT

updateCache = updateCache && !doNotUpdateCacheForMbp;

#endif

if (updateCache)

{

if (!JavascriptProxy::Is(receiver))

{

CacheOperators::CachePropertyWrite(RecyclableObject::FromVar(receiver), isRoot, typeWithoutProperty, propertyId, info, requestContext);

}

}

return TRUE;

}

}

return FALSE;

}

BOOL JavascriptOperators::IsNumberFromNativeArray(Var instance, uint32 index, ScriptContext\* scriptContext)

{

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

Js::TypeId instanceType = JavascriptOperators::GetTypeId(instance);

// Fast path for native and typed arrays.

if ( (instanceType == TypeIds\_NativeIntArray || instanceType == TypeIds\_NativeFloatArray) || (instanceType >= TypeIds\_Int8Array && instanceType <= TypeIds\_Uint64Array) )

{

RecyclableObject\* object = RecyclableObject::FromVar(instance);

Var member;

// If the item is found in the array own body, then it is a number

if (JavascriptOperators::GetOwnItem(object, index, &member, scriptContext))

return TRUE;

}

return FALSE;

}

BOOL JavascriptOperators::GetAccessors(RecyclableObject\* instance, PropertyId propertyId, ScriptContext\* requestContext, Var\* getter, Var\* setter)

{

RecyclableObject\* object = instance;

while (JavascriptOperators::GetTypeId(object) != TypeIds\_Null)

{

if (object->GetAccessors(propertyId, getter, setter, requestContext))

{

\*getter = JavascriptOperators::CanonicalizeAccessor(\*getter, requestContext);

\*setter = JavascriptOperators::CanonicalizeAccessor(\*setter, requestContext);

return TRUE;

}

if (object->SkipsPrototype())

{

break;

}

object = JavascriptOperators::GetPrototypeNoTrap(object);

}

return FALSE;

}

BOOL JavascriptOperators::SetAccessors(RecyclableObject\* instance, PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags)

{

BOOL result = instance && instance->SetAccessors(propertyId, getter, setter, flags);

return result;

}

BOOL JavascriptOperators::OP\_SetProperty(Var instance, PropertyId propertyId, Var newValue, ScriptContext\* scriptContext, PropertyValueInfo \* info, PropertyOperationFlags flags, Var thisInstance)

{

// The call into ToObject(dynamicObject) is avoided here by checking for null and undefined and doing nothing when dynamicObject is a primitive value.

if (thisInstance == nullptr)

{

thisInstance = instance;

}

TypeId typeId = JavascriptOperators::GetTypeId(thisInstance);

if (typeId == TypeIds\_Null || typeId == TypeIds\_Undefined)

{

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotSet\_NullOrUndefined, scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

return TRUE;

}

else if (typeId == TypeIds\_VariantDate)

{

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_VarDate, scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

return TRUE;

}

if (!TaggedNumber::Is(thisInstance))

{

return JavascriptOperators::SetProperty(RecyclableObject::FromVar(thisInstance), RecyclableObject::FromVar(instance), propertyId, newValue, info, scriptContext, flags);

}

JavascriptError::ThrowCantAssignIfStrictMode(flags, scriptContext);

return false;

}

BOOL JavascriptOperators::OP\_StFunctionExpression(Var obj, PropertyId propertyId, Var newValue)

{

RecyclableObject\* instance = RecyclableObject::FromVar(obj);

instance->SetProperty(propertyId, newValue, PropertyOperation\_None, NULL);

instance->SetWritable(propertyId, FALSE);

instance->SetConfigurable(propertyId, FALSE);

return TRUE;

}

BOOL JavascriptOperators::OP\_InitClassMember(Var obj, PropertyId propertyId, Var newValue)

{

RecyclableObject\* instance = RecyclableObject::FromVar(obj);

PropertyOperationFlags flags = PropertyOperation\_None;

PropertyAttributes attributes = PropertyClassMemberDefaults;

instance->SetPropertyWithAttributes(propertyId, newValue, attributes, NULL, flags);

return TRUE;

}

BOOL JavascriptOperators::OP\_InitLetProperty(Var obj, PropertyId propertyId, Var newValue)

{

RecyclableObject\* instance = RecyclableObject::FromVar(obj);

PropertyOperationFlags flags = instance->GetScriptContext()->IsUndeclBlockVar(newValue) ? PropertyOperation\_SpecialValue : PropertyOperation\_None;

PropertyAttributes attributes = PropertyLetDefaults;

if (RootObjectBase::Is(instance))

{

attributes |= PropertyLetConstGlobal;

}

instance->SetPropertyWithAttributes(propertyId, newValue, attributes, NULL, (PropertyOperationFlags)(flags | PropertyOperation\_AllowUndecl));

return TRUE;

}

BOOL JavascriptOperators::OP\_InitConstProperty(Var obj, PropertyId propertyId, Var newValue)

{

RecyclableObject\* instance = RecyclableObject::FromVar(obj);

PropertyOperationFlags flags = instance->GetScriptContext()->IsUndeclBlockVar(newValue) ? PropertyOperation\_SpecialValue : PropertyOperation\_None;

PropertyAttributes attributes = PropertyConstDefaults;

if (RootObjectBase::Is(instance))

{

attributes |= PropertyLetConstGlobal;

}

instance->SetPropertyWithAttributes(propertyId, newValue, attributes, NULL, (PropertyOperationFlags)(flags | PropertyOperation\_AllowUndecl));

return TRUE;

}

BOOL JavascriptOperators::OP\_InitUndeclRootLetProperty(Var obj, PropertyId propertyId)

{

RecyclableObject\* instance = RecyclableObject::FromVar(obj);

PropertyOperationFlags flags = static\_cast<PropertyOperationFlags>(PropertyOperation\_SpecialValue | PropertyOperation\_AllowUndecl);

PropertyAttributes attributes = PropertyLetDefaults | PropertyLetConstGlobal;

instance->SetPropertyWithAttributes(propertyId, instance->GetLibrary()->GetUndeclBlockVar(), attributes, NULL, flags);

return TRUE;

}

BOOL JavascriptOperators::OP\_InitUndeclRootConstProperty(Var obj, PropertyId propertyId)

{

RecyclableObject\* instance = RecyclableObject::FromVar(obj);

PropertyOperationFlags flags = static\_cast<PropertyOperationFlags>(PropertyOperation\_SpecialValue | PropertyOperation\_AllowUndecl);

PropertyAttributes attributes = PropertyConstDefaults | PropertyLetConstGlobal;

instance->SetPropertyWithAttributes(propertyId, instance->GetLibrary()->GetUndeclBlockVar(), attributes, NULL, flags);

return TRUE;

}

BOOL JavascriptOperators::OP\_InitUndeclConsoleLetProperty(Var obj, PropertyId propertyId)

{

FrameDisplay \*pScope = (FrameDisplay\*)obj;

AssertMsg(ConsoleScopeActivationObject::Is((DynamicObject\*)pScope->GetItem(pScope->GetLength() - 1)), "How come we got this opcode without ConsoleScopeActivationObject?");

RecyclableObject\* instance = RecyclableObject::FromVar(pScope->GetItem(0));

PropertyOperationFlags flags = static\_cast<PropertyOperationFlags>(PropertyOperation\_SpecialValue | PropertyOperation\_AllowUndecl);

PropertyAttributes attributes = PropertyLetDefaults;

instance->SetPropertyWithAttributes(propertyId, instance->GetLibrary()->GetUndeclBlockVar(), attributes, NULL, flags);

return TRUE;

}

BOOL JavascriptOperators::OP\_InitUndeclConsoleConstProperty(Var obj, PropertyId propertyId)

{

FrameDisplay \*pScope = (FrameDisplay\*)obj;

AssertMsg(ConsoleScopeActivationObject::Is((DynamicObject\*)pScope->GetItem(pScope->GetLength() - 1)), "How come we got this opcode without ConsoleScopeActivationObject?");

RecyclableObject\* instance = RecyclableObject::FromVar(pScope->GetItem(0));

PropertyOperationFlags flags = static\_cast<PropertyOperationFlags>(PropertyOperation\_SpecialValue | PropertyOperation\_AllowUndecl);

PropertyAttributes attributes = PropertyConstDefaults;

instance->SetPropertyWithAttributes(propertyId, instance->GetLibrary()->GetUndeclBlockVar(), attributes, NULL, flags);

return TRUE;

}

BOOL JavascriptOperators::InitProperty(RecyclableObject\* instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags)

{

return instance && instance->InitProperty(propertyId, newValue, flags);

}

BOOL JavascriptOperators::OP\_InitProperty(Var instance, PropertyId propertyId, Var newValue)

{

if(TaggedNumber::Is(instance)) { return false; }

return JavascriptOperators::InitProperty(RecyclableObject::FromVar(instance), propertyId, newValue);

}

BOOL JavascriptOperators::DeleteProperty(RecyclableObject\* instance, PropertyId propertyId, PropertyOperationFlags propertyOperationFlags)

{

return DeleteProperty\_Impl<false>(instance, propertyId, propertyOperationFlags);

}

BOOL JavascriptOperators::DeletePropertyUnscopables(RecyclableObject\* instance, PropertyId propertyId, PropertyOperationFlags propertyOperationFlags)

{

return DeleteProperty\_Impl<true>(instance, propertyId, propertyOperationFlags);

}

template<bool unscopables>

BOOL JavascriptOperators::DeleteProperty\_Impl(RecyclableObject\* instance, PropertyId propertyId, PropertyOperationFlags propertyOperationFlags)

{

if (unscopables && JavascriptOperators::IsPropertyUnscopable(instance, propertyId))

{

return false;

}

#ifdef ENABLE\_MUTATION\_BREAKPOINT

ScriptContext \*scriptContext = instance->GetScriptContext();

if (MutationBreakpoint::IsFeatureEnabled(scriptContext)

&& scriptContext->HasMutationBreakpoints())

{

MutationBreakpoint::HandleDeleteProperty(scriptContext, instance, propertyId);

}

#endif

// !unscopables will hit the return statement on the first iteration

return instance->DeleteProperty(propertyId, propertyOperationFlags);

}

Var JavascriptOperators::OP\_DeleteProperty(Var instance, PropertyId propertyId, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags)

{

if(TaggedNumber::Is(instance))

{

return scriptContext->GetLibrary()->GetTrue();

}

TypeId typeId = JavascriptOperators::GetTypeId(instance);

if (typeId == TypeIds\_Null || typeId == TypeIds\_Undefined)

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotDelete\_NullOrUndefined,

scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

RecyclableObject \*recyclableObject = RecyclableObject::FromVar(instance);

return scriptContext->GetLibrary()->CreateBoolean(

JavascriptOperators::DeleteProperty(recyclableObject, propertyId, propertyOperationFlags));

}

Var JavascriptOperators::OP\_DeleteRootProperty(Var instance, PropertyId propertyId, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags)

{

AssertMsg(RootObjectBase::Is(instance), "Root must be a global object!");

RootObjectBase\* rootObject = static\_cast<RootObjectBase\*>(instance);

return scriptContext->GetLibrary()->CreateBoolean(

rootObject->DeleteRootProperty(propertyId, propertyOperationFlags));

}

template <bool IsFromFullJit, class TInlineCache>

\_\_inline void JavascriptOperators::PatchSetPropertyScoped(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var newValue, Var defaultInstance, PropertyOperationFlags propertyOperationFlags)

{

// Set the property using a scope stack rather than an individual instance.

// Walk the stack until we find an instance that has the property and store

// the new value there.

//

// To propagate 'this' pointer, walk up the stack and update scopes

// where field '\_lexicalThisSlotSymbol' exists and stop at the

// scope where field '\_lexicalNewTargetSymbol' also exists, which

// indicates class constructor.

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

uint16 length = pDisplay->GetLength();

DynamicObject \*object;

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

bool allowUndecInConsoleScope = (propertyOperationFlags & PropertyOperation\_AllowUndeclInConsoleScope) == PropertyOperation\_AllowUndeclInConsoleScope;

bool isLexicalThisSlotSymbol = (propertyId == PropertyIds::\_lexicalThisSlotSymbol);

for (uint16 i = 0; i < length; i++)

{

object = (DynamicObject\*)pDisplay->GetItem(i);

AssertMsg(!ConsoleScopeActivationObject::Is(object) || (i == length - 1), "Invalid location for ConsoleScopeActivationObject");

Type\* type = object->GetType();

if (CacheOperators::TrySetProperty<true, true, true, true, true, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, false, propertyId, newValue, scriptContext, propertyOperationFlags, nullptr, &info))

{

if (isLexicalThisSlotSymbol && !JavascriptOperators::HasProperty(object, PropertyIds::\_lexicalNewTargetSymbol))

{

continue;

}

return;

}

// In scoped set property, we need to set the property when it is available; it could be a setter

// or normal property. we need to check setter first, and if no setter is available, but HasProperty

// is true, this must be a normal property.

// TODO: merge OP\_HasProperty and GetSetter in one pass if there is perf problem. In fastDOM we have quite

// a lot of setters so separating the two might be actually faster.

Var setterValueOrProxy = nullptr;

DescriptorFlags flags = None;

if (JavascriptOperators::CheckPrototypesForAccessorOrNonWritableProperty(object, propertyId, &setterValueOrProxy, &flags, &info, scriptContext))

{

if ((flags & Accessor) == Accessor)

{

if (setterValueOrProxy)

{

JavascriptFunction\* func = (JavascriptFunction\*)setterValueOrProxy;

Assert(info.GetFlags() == InlineCacheSetterFlag || info.GetPropertyIndex() == Constants::NoSlot);

CacheOperators::CachePropertyWrite(object, false, type, propertyId, &info, scriptContext);

JavascriptOperators::CallSetter(func, object, newValue, scriptContext);

}

Assert(!isLexicalThisSlotSymbol);

return;

}

else if ((flags & Proxy) == Proxy)

{

Assert(JavascriptProxy::Is(setterValueOrProxy));

JavascriptProxy\* proxy = JavascriptProxy::FromVar(setterValueOrProxy);

auto fn = [&](RecyclableObject\* target) -> BOOL {

return JavascriptOperators::SetProperty(object, target, propertyId, newValue, scriptContext, propertyOperationFlags);

};

// We can't cache the property at this time. both target and handler can be changed outside of the proxy, so the inline cache needs to be

// invalidate when target, handler, or handler prototype has changed. We don't have a way to achieve this yet.

PropertyValueInfo::SetNoCache(&info, proxy);

PropertyValueInfo::DisablePrototypeCache(&info, proxy); // We can't cache prototype property either

proxy->SetPropertyTrap(object, JavascriptProxy::SetPropertyTrapKind::SetPropertyKind, propertyId, newValue, scriptContext);

}

else

{

Assert((flags & Data) == Data && (flags & Writable) == None);

if (!allowUndecInConsoleScope)

{

if (flags & Const)

{

JavascriptError::ThrowReferenceError(scriptContext, ERRAssignmentToConst);

}

Assert(!isLexicalThisSlotSymbol);

return;

}

}

}

else if (!JavascriptOperators::IsObject(object))

{

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, scriptContext);

}

// Need to do a "get" of the current value (if any) to make sure that we're not writing to

// let/const before declaration, but we need to disable implicit calls around the "get",

// so we need to do a "has" first to make sure the "get" is valid (e.g., "get" on a HostDispatch

// with implicit calls disabled will always "succeed").

if (JavascriptOperators::HasProperty(object, propertyId))

{

if (scriptContext->GetConfig()->IsLetAndConstEnabled())

{

DisableImplicitFlags disableImplicitFlags =

scriptContext->GetThreadContext()->GetDisableImplicitFlags();

scriptContext->GetThreadContext()->SetDisableImplicitFlags(DisableImplicitCallAndExceptionFlag);

Var value;

BOOL result = JavascriptOperators::GetProperty(object, propertyId, &value, scriptContext, nullptr);

scriptContext->GetThreadContext()->SetDisableImplicitFlags(disableImplicitFlags);

if (result && scriptContext->IsUndeclBlockVar(value) && !allowUndecInConsoleScope && !isLexicalThisSlotSymbol)

{

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UseBeforeDeclaration);

}

}

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

PropertyOperationFlags setPropertyOpFlags = allowUndecInConsoleScope ? PropertyOperation\_AllowUndeclInConsoleScope : PropertyOperation\_None;

object->SetProperty(propertyId, newValue, setPropertyOpFlags, &info);

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchSetPropertyScoped", propertyId, scriptContext, object);

}

#endif

if (!JavascriptProxy::Is(object) && !allowUndecInConsoleScope)

{

CacheOperators::CachePropertyWrite(object, false, type, propertyId, &info, scriptContext);

}

if (isLexicalThisSlotSymbol && !JavascriptOperators::HasProperty(object, PropertyIds::\_lexicalNewTargetSymbol))

{

continue;

}

return;

}

}

Assert(!isLexicalThisSlotSymbol);

// If we have console scope and no one in the scope had the property add it to console scope

if ((length > 0) && ConsoleScopeActivationObject::Is(pDisplay->GetItem(length - 1)))

{

RecyclableObject\* obj = RecyclableObject::FromVar((DynamicObject\*)pDisplay->GetItem(length - 1));

OUTPUT\_TRACE(Js::ConsoleScopePhase, L"Adding property '%s' to console scope object\n", scriptContext->GetPropertyName(propertyId)->GetBuffer());

JavascriptOperators::SetProperty(obj, obj, propertyId, newValue, scriptContext, propertyOperationFlags);

return;

}

// No one in the scope stack has the property, so add it to the default instance provided by the caller.

AssertMsg(!TaggedNumber::Is(defaultInstance), "Root object is an int or tagged float?");

Assert(defaultInstance != nullptr);

RecyclableObject\* obj = RecyclableObject::FromVar(defaultInstance);

{

//SetPropertyScoped does not use inline cache for default instance

PropertyValueInfo info;

JavascriptOperators::SetRootProperty(obj, propertyId, newValue, &info, scriptContext, (PropertyOperationFlags)(propertyOperationFlags | PropertyOperation\_Root));

}

}

template void JavascriptOperators::PatchSetPropertyScoped<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var newValue, Var defaultInstance, PropertyOperationFlags propertyOperationFlags);

template void JavascriptOperators::PatchSetPropertyScoped<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var newValue, Var defaultInstance, PropertyOperationFlags propertyOperationFlags);

template void JavascriptOperators::PatchSetPropertyScoped<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var newValue, Var defaultInstance, PropertyOperationFlags propertyOperationFlags);

template void JavascriptOperators::PatchSetPropertyScoped<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var newValue, Var defaultInstance, PropertyOperationFlags propertyOperationFlags);

BOOL JavascriptOperators::OP\_InitFuncScoped(FrameDisplay \*pScope, PropertyId propertyId, Var newValue, Var defaultInstance, ScriptContext\* scriptContext)

{

int i;

int length = pScope->GetLength();

DynamicObject \*obj;

for (i = 0; i < length; i++)

{

obj = (DynamicObject\*)pScope->GetItem(i);

if (obj->InitFuncScoped(propertyId, newValue))

{

return TRUE;

}

}

AssertMsg(!TaggedNumber::Is(defaultInstance), "Root object is an int or tagged float?");

return RecyclableObject::FromVar(defaultInstance)->InitFuncScoped(propertyId, newValue);

}

BOOL JavascriptOperators::OP\_InitPropertyScoped(FrameDisplay \*pScope, PropertyId propertyId, Var newValue, Var defaultInstance, ScriptContext\* scriptContext)

{

int i;

int length = pScope->GetLength();

DynamicObject \*obj;

for (i = 0; i < length; i++)

{

obj = (DynamicObject\*)pScope->GetItem(i);

if (obj->InitPropertyScoped(propertyId, newValue))

{

return TRUE;

}

}

AssertMsg(!TaggedNumber::Is(defaultInstance), "Root object is an int or tagged float?");

return RecyclableObject::FromVar(defaultInstance)->InitPropertyScoped(propertyId, newValue);

}

Var JavascriptOperators::OP\_DeletePropertyScoped(

FrameDisplay \*pScope,

PropertyId propertyId,

Var defaultInstance,

ScriptContext\* scriptContext,

PropertyOperationFlags propertyOperationFlags)

{

int i;

int length = pScope->GetLength();

for (i = 0; i < length; i++)

{

DynamicObject \*obj = (DynamicObject\*)pScope->GetItem(i);

if (JavascriptOperators::HasProperty(obj, propertyId))

{

return scriptContext->GetLibrary()->CreateBoolean(JavascriptOperators::DeleteProperty(obj, propertyId, propertyOperationFlags));

}

}

return JavascriptOperators::OP\_DeleteRootProperty(RecyclableObject::FromVar(defaultInstance), propertyId, scriptContext, propertyOperationFlags);

}

Var JavascriptOperators::OP\_TypeofPropertyScoped(FrameDisplay \*pScope, PropertyId propertyId, Var defaultInstance, ScriptContext\* scriptContext)

{

int i;

int length = pScope->GetLength();

for (i = 0; i < length; i++)

{

DynamicObject \*obj = (DynamicObject\*)pScope->GetItem(i);

if (JavascriptOperators::HasProperty(obj, propertyId))

{

return JavascriptOperators::TypeofFld(obj, propertyId, scriptContext);

}

}

return JavascriptOperators::TypeofRootFld(RecyclableObject::FromVar(defaultInstance), propertyId, scriptContext);

}

BOOL JavascriptOperators::HasOwnItem(RecyclableObject\* object, uint32 index)

{

return object->HasOwnItem(index);

}

BOOL JavascriptOperators::HasItem(RecyclableObject\* object, uint64 index)

{

PropertyRecord const \* propertyRecord;

ScriptContext\* scriptContext = object->GetScriptContext();

JavascriptOperators::GetPropertyIdForInt(index, scriptContext, &propertyRecord);

return JavascriptOperators::HasProperty(object, propertyRecord->GetPropertyId());

}

BOOL JavascriptOperators::HasItem(RecyclableObject\* object, uint32 index)

{

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(object);

#endif

while (JavascriptOperators::GetTypeId(object) != TypeIds\_Null)

{

if (object->HasItem(index))

{

return true;

}

// CONSIDER: Numeric property values shouldn't be on the prototype for now but if this changes

// we should add SkipsPrototype support here as well

object = JavascriptOperators::GetPrototypeNoTrap(object);

}

return false;

}

BOOL JavascriptOperators::GetOwnItem(RecyclableObject\* object, uint32 index, Var\* value, ScriptContext\* requestContext)

{

return object->GetItem(object, index, value, requestContext);

}

BOOL JavascriptOperators::GetItem(Var instance, RecyclableObject\* propertyObject, uint32 index, Var\* value, ScriptContext\* requestContext)

{

RecyclableObject\* object = propertyObject;

while (JavascriptOperators::GetTypeId(object) != TypeIds\_Null)

{

if (object->GetItem(instance, index, value, requestContext))

{

return true;

}

if (object->SkipsPrototype())

{

break;

}

object = JavascriptOperators::GetPrototypeNoTrap(object);

}

return false;

}

BOOL JavascriptOperators::GetItemReference(Var instance, RecyclableObject\* propertyObject, uint32 index, Var\* value, ScriptContext\* requestContext)

{

RecyclableObject\* object = propertyObject;

while (JavascriptOperators::GetTypeId(object) != TypeIds\_Null)

{

if (object->GetItemReference(instance, index, value, requestContext))

{

return true;

}

if (object->SkipsPrototype())

{

break;

}

object = JavascriptOperators::GetPrototypeNoTrap(object);

}

return false;

}

BOOL JavascriptOperators::SetItem(Var receiver, RecyclableObject\* object, uint64 index, Var value, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags)

{

PropertyRecord const \* propertyRecord;

JavascriptOperators::GetPropertyIdForInt(index, scriptContext, &propertyRecord);

return JavascriptOperators::SetProperty(receiver, object, propertyRecord->GetPropertyId(), value, scriptContext, propertyOperationFlags);

}

BOOL JavascriptOperators::SetItem(Var receiver, RecyclableObject\* object, uint32 index, Var value, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags, BOOL skipPrototypeCheck /\* = FALSE \*/)

{

Var setterValueOrProxy = nullptr;

DescriptorFlags flags = None;

Assert(!TaggedNumber::Is(receiver));

if (JavascriptOperators::CheckPrototypesForAccessorOrNonWritableItem(object, index, &setterValueOrProxy, &flags, scriptContext, skipPrototypeCheck))

{

scriptContext->GetThreadContext()->AddImplicitCallFlags(ImplicitCall\_NoOpSet);

if ((flags & Accessor) == Accessor)

{

if (JavascriptError::ThrowIfStrictModeUndefinedSetter(propertyOperationFlags, setterValueOrProxy, scriptContext) ||

JavascriptError::ThrowIfNotExtensibleUndefinedSetter(propertyOperationFlags, setterValueOrProxy, scriptContext))

{

return TRUE;

}

if (setterValueOrProxy)

{

RecyclableObject\* func = RecyclableObject::FromVar(setterValueOrProxy);

JavascriptOperators::CallSetter(func, receiver, value, scriptContext);

}

return TRUE;

}

else if ((flags & Proxy) == Proxy)

{

Assert(JavascriptProxy::Is(setterValueOrProxy));

JavascriptProxy\* proxy = JavascriptProxy::FromVar(setterValueOrProxy);

const PropertyRecord\* propertyRecord;

proxy->PropertyIdFromInt(index, &propertyRecord);

return proxy->SetPropertyTrap(receiver, JavascriptProxy::SetPropertyTrapKind::SetItemKind, propertyRecord->GetPropertyId(), value, scriptContext, skipPrototypeCheck);

}

else

{

Assert((flags & Data) == Data && (flags & Writable) == None);

if ((propertyOperationFlags & PropertyOperationFlags::PropertyOperation\_ThrowIfNotExtensible) == PropertyOperationFlags::PropertyOperation\_ThrowIfNotExtensible)

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NonExtensibleObject);

}

JavascriptError::ThrowCantAssign(propertyOperationFlags, scriptContext, index);

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, scriptContext);

return FALSE;

}

}

else if (!JavascriptOperators::IsObject(receiver))

{

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, scriptContext);

return FALSE;

}

return (RecyclableObject::FromVar(receiver))->SetItem(index, value, propertyOperationFlags);

}

BOOL JavascriptOperators::DeleteItem(RecyclableObject\* object, uint32 index, PropertyOperationFlags propertyOperationFlags)

{

return object->DeleteItem(index, propertyOperationFlags);

}

BOOL JavascriptOperators::DeleteItem(RecyclableObject\* object, uint64 index, PropertyOperationFlags propertyOperationFlags)

{

PropertyRecord const \* propertyRecord;

JavascriptOperators::GetPropertyIdForInt(index, object->GetScriptContext(), &propertyRecord);

return JavascriptOperators::DeleteProperty(object, propertyRecord->GetPropertyId(), propertyOperationFlags);

}

BOOL JavascriptOperators::OP\_HasItem(Var instance, Var index, ScriptContext\* scriptContext)

{

RecyclableObject\* object = TaggedNumber::Is(instance) ?

scriptContext->GetLibrary()->GetNumberPrototype() :

RecyclableObject::FromVar(instance);

uint32 indexVal;

PropertyRecord const \* propertyRecord;

bool createIfNotFound = (DynamicType::Is(object->GetTypeId()) &&

static\_cast<DynamicObject\*>(object)->GetTypeHandler()->IsStringTypeHandler()) ||

JavascriptProxy::Is(object);

if (GetIndexType(index, scriptContext, &indexVal, &propertyRecord, createIfNotFound) == IndexType\_Number)

{

return HasItem(object, indexVal);

}

else if (propertyRecord == nullptr)

{

Assert(IsJsNativeObject(object));

#if DBG

JavascriptString\* indexStr = JavascriptConversion::ToString(index, scriptContext);

PropertyRecord const \* debugPropertyRecord;

scriptContext->GetOrAddPropertyRecord(indexStr->GetString(), indexStr->GetLength(), &debugPropertyRecord);

AssertMsg(!JavascriptOperators::HasProperty(object, debugPropertyRecord->GetPropertyId()), "how did this property come? See OS Bug 2727708 if you see this come from the web");

#endif

return FALSE;

}

else

{

return HasProperty(object, propertyRecord->GetPropertyId());

}

}

#if ENABLE\_PROFILE\_INFO

void JavascriptOperators::UpdateNativeArrayProfileInfoToCreateVarArray(Var instance, const bool expectingNativeFloatArray, const bool expectingVarArray)

{

Assert(instance);

Assert(expectingNativeFloatArray ^ expectingVarArray);

if (!JavascriptNativeArray::Is(instance))

{

return;

}

ArrayCallSiteInfo \*const arrayCallSiteInfo = JavascriptNativeArray::FromVar(instance)->GetArrayCallSiteInfo();

if (!arrayCallSiteInfo)

{

return;

}

if (expectingNativeFloatArray)

{

// Profile data is expecting a native float array. Ensure that at the array's creation site, that a native int array

// is not created, such that the profiled array type would be correct.

arrayCallSiteInfo->SetIsNotNativeIntArray();

}

else

{

// Profile data is expecting a var array. Ensure that at the array's creation site, that a native array is not

// created, such that the profiled array type would be correct.

Assert(expectingVarArray);

arrayCallSiteInfo->SetIsNotNativeArray();

}

}

bool JavascriptOperators::SetElementMayHaveImplicitCalls(ScriptContext \*const scriptContext)

{

return

scriptContext->optimizationOverrides.GetArraySetElementFastPathVtable() ==

ScriptContextOptimizationOverrideInfo::InvalidVtable;

}

#endif

RecyclableObject \*JavascriptOperators::GetCallableObjectOrThrow(const Var callee, ScriptContext \*const scriptContext)

{

Assert(callee);

Assert(scriptContext);

if (TaggedNumber::Is(callee))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedFunction /\* TODO-ERROR: get arg name - aFunc \*/);

}

return RecyclableObject::FromVar(callee);

}

#if ENABLE\_NATIVE\_CODEGEN

Var JavascriptOperators::OP\_GetElementI\_JIT(Var instance, Var index, ScriptContext \*scriptContext)

{

Assert(Js::JavascriptStackWalker::ValidateTopJitFrame(scriptContext));

return OP\_GetElementI(instance, index, scriptContext);

}

#else

Var JavascriptOperators::OP\_GetElementI\_JIT(Var instance, Var index, ScriptContext \*scriptContext)

{

return OP\_GetElementI(instance, index, scriptContext);

}

#endif

#if ENABLE\_NATIVE\_CODEGEN

Var JavascriptOperators::OP\_GetElementI\_JIT\_ExpectingNativeFloatArray(Var instance, Var index, ScriptContext \*scriptContext)

{

Assert(Js::JavascriptStackWalker::ValidateTopJitFrame(scriptContext));

UpdateNativeArrayProfileInfoToCreateVarArray(instance, true, false);

return OP\_GetElementI\_JIT(instance, index, scriptContext);

}

Var JavascriptOperators::OP\_GetElementI\_JIT\_ExpectingVarArray(Var instance, Var index, ScriptContext \*scriptContext)

{

Assert(Js::JavascriptStackWalker::ValidateTopJitFrame(scriptContext));

UpdateNativeArrayProfileInfoToCreateVarArray(instance, false, true);

return OP\_GetElementI\_JIT(instance, index, scriptContext);

}

#endif

Var JavascriptOperators::OP\_GetElementI\_UInt32(Var instance, uint32 index, ScriptContext\* scriptContext)

{

#if FLOATVAR

return OP\_GetElementI\_JIT(instance, Js::JavascriptNumber::ToVar(index, scriptContext), scriptContext);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_GetElementI\_JIT(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer), scriptContext);

#endif

}

Var JavascriptOperators::OP\_GetElementI\_UInt32\_ExpectingNativeFloatArray(Var instance, uint32 index, ScriptContext\* scriptContext)

{

#if ENABLE\_PROFILE\_INFO

UpdateNativeArrayProfileInfoToCreateVarArray(instance, true, false);

#endif

return OP\_GetElementI\_UInt32(instance, index, scriptContext);

}

Var JavascriptOperators::OP\_GetElementI\_UInt32\_ExpectingVarArray(Var instance, uint32 index, ScriptContext\* scriptContext)

{

#if ENABLE\_PROFILE\_INFO

UpdateNativeArrayProfileInfoToCreateVarArray(instance, false, true);

#endif

return OP\_GetElementI\_UInt32(instance, index, scriptContext);

}

Var JavascriptOperators::OP\_GetElementI\_Int32(Var instance, int32 index, ScriptContext\* scriptContext)

{

#if FLOATVAR

return OP\_GetElementI\_JIT(instance, Js::JavascriptNumber::ToVar(index, scriptContext), scriptContext);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_GetElementI\_JIT(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer), scriptContext);

#endif

}

Var JavascriptOperators::OP\_GetElementI\_Int32\_ExpectingNativeFloatArray(Var instance, int32 index, ScriptContext\* scriptContext)

{

#if ENABLE\_PROFILE\_INFO

UpdateNativeArrayProfileInfoToCreateVarArray(instance, true, false);

#endif

return OP\_GetElementI\_Int32(instance, index, scriptContext);

}

Var JavascriptOperators::OP\_GetElementI\_Int32\_ExpectingVarArray(Var instance, int32 index, ScriptContext\* scriptContext)

{

#if ENABLE\_PROFILE\_INFO

UpdateNativeArrayProfileInfoToCreateVarArray(instance, false, true);

#endif

return OP\_GetElementI\_Int32(instance, index, scriptContext);

}

BOOL JavascriptOperators::GetItemFromArrayPrototype(JavascriptArray \* arr, int32 indexInt, Var \* result, ScriptContext \* scriptContext)

{

// try get from Array prototype

RecyclableObject\* prototype = arr->GetPrototype();

if (JavascriptOperators::GetTypeId(prototype) != TypeIds\_Array) //This can be TypeIds\_ES5Array (or any other object changed through \_\_proto\_\_).

{

return false;

}

JavascriptArray\* arrayPrototype = JavascriptArray::FromVar(prototype); //Prototype must be Array.prototype (unless changed through \_\_proto\_\_)

AssertMsg(scriptContext->GetConfig()->Is\_\_proto\_\_Enabled()

|| arrayPrototype->GetScriptContext()->GetLibrary()->GetArrayPrototype() == arrayPrototype, "This function is supported only for [[class]] Array");

if (arrayPrototype->GetLength() && arrayPrototype->GetItem(arrayPrototype, (uint32)indexInt, result, scriptContext))

{

return true;

}

prototype = arrayPrototype->GetPrototype(); //Its prototype must be Object.prototype (unless changed through \_\_proto\_\_)

AssertMsg(scriptContext->GetConfig()->Is\_\_proto\_\_Enabled()

|| prototype->GetScriptContext()->GetLibrary()->GetObjectPrototype() == prototype, "This function is supported only for [[class]] Array");

if (prototype->GetScriptContext()->GetLibrary()->GetObjectPrototype() != prototype)

{

return false;

}

if (DynamicObject::FromVar(prototype)->HasNonEmptyObjectArray())

{

if (prototype->GetItem(arr, (uint32)indexInt, result, scriptContext))

{

return true;

}

}

\*result = scriptContext->GetMissingItemResult(arr, indexInt);

return true;

}

template <typename T>

BOOL JavascriptOperators::OP\_GetElementI\_ArrayFastPath(T \* arr, int indexInt, Var \* result, ScriptContext \* scriptContext)

{

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(arr);

#endif

if (indexInt >= 0)

{

if (!CrossSite::IsCrossSiteObjectTyped(arr))

{

if (arr->T::DirectGetVarItemAt((uint32)indexInt, result, scriptContext))

{

return true;

}

}

else

{

if (arr->GetItem(arr, (uint32)indexInt, result, scriptContext))

{

return true;

}

}

return GetItemFromArrayPrototype(arr, indexInt, result, scriptContext);

}

return false;

}

Var JavascriptOperators::OP\_GetElementI(Var instance, Var index, ScriptContext\* scriptContext)

{

JavascriptString \*temp = NULL;

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

if (TaggedInt::Is(index))

{

TaggedIntIndex:

switch (JavascriptOperators::GetTypeId(instance))

{

case TypeIds\_Array: //fast path for array

{

Var result;

if (OP\_GetElementI\_ArrayFastPath(JavascriptArray::FromVar(instance), TaggedInt::ToInt32(index), &result, scriptContext))

{

return result;

}

break;

}

case TypeIds\_NativeIntArray:

{

Var result;

if (OP\_GetElementI\_ArrayFastPath(JavascriptNativeIntArray::FromVar(instance), TaggedInt::ToInt32(index), &result, scriptContext))

{

return result;

}

break;

}

case TypeIds\_NativeFloatArray:

{

Var result;

if (OP\_GetElementI\_ArrayFastPath(JavascriptNativeFloatArray::FromVar(instance), TaggedInt::ToInt32(index), &result, scriptContext))

{

return result;

}

break;

}

case TypeIds\_String: // fast path for string

{

charcount\_t indexInt = TaggedInt::ToUInt32(index);

JavascriptString\* string = JavascriptString::FromVar(instance);

Var result;

if (string->JavascriptString::GetItem(instance, indexInt, &result, scriptContext))

{

return result;

}

break;

}

case TypeIds\_Int8Array:

{

// The typed array will deal with all possible values for the index

int32 indexInt = TaggedInt::ToInt32(index);

if (VirtualTableInfo<Int8VirtualArray>::HasVirtualTable(instance))

{

Int8VirtualArray\* int8Array = Int8VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int8Array) && indexInt >= 0)

{

return int8Array->DirectGetItem(indexInt);

}

}

else

{

Int8Array\* int8Array = Int8Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int8Array) && indexInt >= 0)

{

return int8Array->DirectGetItem(indexInt);

}

}

break;

}

case TypeIds\_Uint8Array:

{

// The typed array will deal with all possible values for the index

int32 indexInt = TaggedInt::ToInt32(index);

if (VirtualTableInfo<Uint8VirtualArray>::HasVirtualTable(instance))

{

Uint8VirtualArray\* uint8Array = Uint8VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint8Array) && indexInt >= 0)

{

return uint8Array->DirectGetItem(indexInt);

}

}

else

{

Uint8Array\* uint8Array = Uint8Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint8Array) && indexInt >= 0)

{

return uint8Array->DirectGetItem(indexInt);

}

}

break;

}

case TypeIds\_Uint8ClampedArray:

{

// The typed array will deal with all possible values for the index

int32 indexInt = TaggedInt::ToInt32(index);

if (VirtualTableInfo<Uint8ClampedVirtualArray>::HasVirtualTable(instance))

{

Uint8ClampedVirtualArray\* uint8ClampedArray = Uint8ClampedVirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint8ClampedArray) && indexInt >= 0)

{

return uint8ClampedArray->DirectGetItem(indexInt);

}

}

else

{

Uint8ClampedArray\* uint8ClampedArray = Uint8ClampedArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint8ClampedArray) && indexInt >= 0)

{

return uint8ClampedArray->DirectGetItem(indexInt);

}

}

break;

}

case TypeIds\_Int16Array:

{

// The type array will deal with all possible values for the index

int32 indexInt = TaggedInt::ToInt32(index);

if (VirtualTableInfo<Int16VirtualArray>::HasVirtualTable(instance))

{

Int16VirtualArray\* int16Array = Int16VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int16Array) && indexInt >= 0)

{

return int16Array->DirectGetItem(indexInt);

}

}

else

{

Int16Array\* int16Array = Int16Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int16Array) && indexInt >= 0)

{

return int16Array->DirectGetItem(indexInt);

}

}

break;

}

case TypeIds\_Uint16Array:

{

// The type array will deal with all possible values for the index

int32 indexInt = TaggedInt::ToInt32(index);

if (VirtualTableInfo<Uint16VirtualArray>::HasVirtualTable(instance))

{

Uint16VirtualArray\* uint16Array = Uint16VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint16Array) && indexInt >= 0)

{

return uint16Array->DirectGetItem(indexInt);

}

}

else

{

Uint16Array\* uint16Array = Uint16Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint16Array) && indexInt >= 0)

{

return uint16Array->DirectGetItem(indexInt);

}

}

break;

}

case TypeIds\_Int32Array:

{

// The type array will deal with all possible values for the index

int32 indexInt = TaggedInt::ToInt32(index);

if (VirtualTableInfo<Int32VirtualArray>::HasVirtualTable(instance))

{

Int32VirtualArray\* int32Array = Int32VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int32Array) && indexInt >= 0)

{

return int32Array->DirectGetItem(indexInt);

}

}

else

{

Int32Array\* int32Array = Int32Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int32Array) && indexInt >= 0)

{

return int32Array->DirectGetItem(indexInt);

}

}

break;

}

case TypeIds\_Uint32Array:

{

// The type array will deal with all possible values for the index

int32 indexInt = TaggedInt::ToInt32(index);

if (VirtualTableInfo<Uint32VirtualArray>::HasVirtualTable(instance))

{

Uint32VirtualArray\* uint32Array = Uint32VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint32Array) && indexInt >= 0)

{

return uint32Array->DirectGetItem(indexInt);

}

}

else

{

Uint32Array\* uint32Array = Uint32Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint32Array) && indexInt >= 0)

{

return uint32Array->DirectGetItem(indexInt);

}

}

break;

}

case TypeIds\_Float32Array:

{

// The type array will deal with all possible values for the index

int32 indexInt = TaggedInt::ToInt32(index);

if (VirtualTableInfo<Float32VirtualArray>::HasVirtualTable(instance))

{

Float32VirtualArray\* float32Array = Float32VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(float32Array) && indexInt >= 0)

{

return float32Array->DirectGetItem(indexInt);

}

}

else

{

Float32Array\* float32Array = Float32Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(float32Array) && indexInt >= 0)

{

return float32Array->DirectGetItem(indexInt);

}

}

break;

}

case TypeIds\_Float64Array:

{

// The type array will deal with all possible values for the index

int32 indexInt = TaggedInt::ToInt32(index);

if (VirtualTableInfo<Float64VirtualArray>::HasVirtualTable(instance))

{

Float64VirtualArray\* float64Array = Float64VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(float64Array) && indexInt >= 0)

{

return float64Array->DirectGetItem(indexInt);

}

}

else

{

Float64Array\* float64Array = Float64Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(float64Array) && indexInt >= 0)

{

return float64Array->DirectGetItem(indexInt);

}

}

break;

}

default:

break;

}

}

else if (JavascriptNumber::Is\_NoTaggedIntCheck(index))

{

uint32 uint32Index = JavascriptConversion::ToUInt32(index, scriptContext);

if ((double)uint32Index == JavascriptNumber::GetValue(index) && !TaggedInt::IsOverflow(uint32Index))

{

index = TaggedInt::ToVarUnchecked(uint32Index);

goto TaggedIntIndex;

}

}

else if (JavascriptString::Is(index)) // fastpath for PropertyStrings

{

temp = JavascriptString::FromVar(index);

Assert(temp->GetScriptContext() == scriptContext);

if (VirtualTableInfo<Js::PropertyString>::HasVirtualTable(temp))

{

PropertyString \* propertyString = (PropertyString\*)temp;

PropertyCache const \*cache = propertyString->GetPropertyCache();

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined,

JavascriptString::FromVar(index)->GetSz());

}

if (object->GetType() == cache->type)

{

#if DBG\_DUMP

scriptContext->forinCache++;

#endif

Assert(object->GetScriptContext() == scriptContext);

Var value;

if (cache->isInlineSlot)

{

value = DynamicObject::FromVar(object)->GetInlineSlot(cache->dataSlotIndex);

}

else

{

value = DynamicObject::FromVar(object)->GetAuxSlot(cache->dataSlotIndex);

}

Assert(!CrossSite::NeedMarshalVar(value, scriptContext));

Assert(value == JavascriptOperators::GetProperty(object, propertyString->GetPropertyRecord()->GetPropertyId(), scriptContext)

|| value == JavascriptOperators::GetRootProperty(object, propertyString->GetPropertyRecord()->GetPropertyId(), scriptContext));

return value;

}

#if DBG\_DUMP

scriptContext->forinNoCache++;

#endif

PropertyRecord const \* propertyRecord = propertyString->GetPropertyRecord();

Var value;

if (propertyRecord->IsNumeric())

{

if (JavascriptOperators::GetItem(instance, object, propertyRecord->GetNumericValue(), &value, scriptContext))

{

return value;

}

}

else

{

if (JavascriptOperators::GetPropertyWPCache(instance, object, propertyRecord->GetPropertyId(), &value, scriptContext, propertyString))

{

return value;

}

}

return scriptContext->GetLibrary()->GetUndefined();

}

#if DBG\_DUMP

scriptContext->forinNoCache++;

#endif

}

return JavascriptOperators::GetElementIHelper(instance, index, instance, scriptContext);

}

Var JavascriptOperators::GetElementIHelper(Var instance, Var index, Var receiver, ScriptContext\* scriptContext)

{

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined, GetPropertyDisplayNameForError(index, scriptContext));

}

else

{

return scriptContext->GetLibrary()->GetUndefined();

}

}

uint32 indexVal;

PropertyRecord const \* propertyRecord;

JavascriptString \* propertyNameString;

Var value;

bool createIfNotFound = !IsJsNativeObject(object);

IndexType indexType = GetIndexType(index, scriptContext, &indexVal, &propertyRecord, &propertyNameString, createIfNotFound, true);

if (indexType == IndexType\_Number)

{

if (JavascriptOperators::GetItem(receiver, object, indexVal, &value, scriptContext))

{

return value;

}

}

else if (indexType == IndexType\_JavascriptString)

{

if (JavascriptOperators::GetPropertyWPCache(receiver, object, propertyNameString, &value, scriptContext, nullptr))

{

return value;

}

}

else

{

// We called GetIndexType with preferJavascriptString as true, so we mush have a propertyRecord

Assert(indexType == IndexType\_PropertyId);

Assert(propertyRecord);

if (JavascriptOperators::GetPropertyWPCache(receiver, object, propertyRecord->GetPropertyId(), &value, scriptContext, nullptr))

{

return value;

}

}

return scriptContext->GetMissingItemResult(object, indexVal);

}

int32 JavascriptOperators::OP\_GetNativeIntElementI(Var instance, Var index)

{

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

if (TaggedInt::Is(index))

{

int32 indexInt = TaggedInt::ToInt32(index);

if (indexInt < 0)

{

return JavascriptNativeIntArray::MissingItem;

}

JavascriptArray \* arr = JavascriptArray::FromVar(instance);

int32 result;

if (arr->DirectGetItemAt((uint32)indexInt, &result))

{

return result;

}

}

else if (JavascriptNumber::Is\_NoTaggedIntCheck(index))

{

int32 indexInt;

bool isInt32;

double dIndex = JavascriptNumber::GetValue(index);

if (JavascriptNumber::TryGetInt32OrUInt32Value(dIndex, &indexInt, &isInt32))

{

if (isInt32 && indexInt < 0)

{

return JavascriptNativeIntArray::MissingItem;

}

JavascriptArray \* arr = JavascriptArray::FromVar(instance);

int32 result;

if (arr->DirectGetItemAt((uint32)indexInt, &result))

{

return result;

}

}

}

else

{

AssertMsg(false, "Non-numerical index in this helper?");

}

return JavascriptNativeIntArray::MissingItem;

}

int32 JavascriptOperators::OP\_GetNativeIntElementI\_UInt32(Var instance, uint32 index, ScriptContext\* scriptContext)

{

#if FLOATVAR

return OP\_GetNativeIntElementI(instance, Js::JavascriptNumber::ToVar(index, scriptContext));

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_GetNativeIntElementI(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer));

#endif

}

int32 JavascriptOperators::OP\_GetNativeIntElementI\_Int32(Var instance, int32 index, ScriptContext\* scriptContext)

{

#if FLOATVAR

return OP\_GetNativeIntElementI(instance, Js::JavascriptNumber::ToVar(index, scriptContext));

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_GetNativeIntElementI(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer));

#endif

}

double JavascriptOperators::OP\_GetNativeFloatElementI(Var instance, Var index)

{

double result = 0;

if (TaggedInt::Is(index))

{

int32 indexInt = TaggedInt::ToInt32(index);

if (indexInt < 0)

{

result = JavascriptNativeFloatArray::MissingItem;

}

else

{

JavascriptArray \* arr = JavascriptArray::FromVar(instance);

if (!arr->DirectGetItemAt((uint32)indexInt, &result))

{

result = JavascriptNativeFloatArray::MissingItem;

}

}

}

else if (JavascriptNumber::Is\_NoTaggedIntCheck(index))

{

int32 indexInt;

bool isInt32;

double dIndex = JavascriptNumber::GetValue(index);

if (JavascriptNumber::TryGetInt32OrUInt32Value(dIndex, &indexInt, &isInt32))

{

if (isInt32 && indexInt < 0)

{

result = JavascriptNativeFloatArray::MissingItem;

}

else

{

JavascriptArray \* arr = JavascriptArray::FromVar(instance);

if (!arr->DirectGetItemAt((uint32)indexInt, &result))

{

result = JavascriptNativeFloatArray::MissingItem;

}

}

}

}

else

{

AssertMsg(false, "Non-numerical index in this helper?");

}

return result;

}

double JavascriptOperators::OP\_GetNativeFloatElementI\_UInt32(Var instance, uint32 index, ScriptContext\* scriptContext)

{

#if FLOATVAR

return OP\_GetNativeFloatElementI(instance, Js::JavascriptNumber::ToVar(index, scriptContext));

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_GetNativeFloatElementI(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer));

#endif

}

double JavascriptOperators::OP\_GetNativeFloatElementI\_Int32(Var instance, int32 index, ScriptContext\* scriptContext)

{

#if FLOATVAR

return OP\_GetNativeFloatElementI(instance, Js::JavascriptNumber::ToVar(index, scriptContext));

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_GetNativeFloatElementI(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer));

#endif

}

Var JavascriptOperators::OP\_GetMethodElement\_UInt32(Var instance, uint32 index, ScriptContext\* scriptContext)

{

#if FLOATVAR

return OP\_GetMethodElement(instance, Js::JavascriptNumber::ToVar(index, scriptContext), scriptContext);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_GetMethodElement(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer), scriptContext);

#endif

}

Var JavascriptOperators::OP\_GetMethodElement\_Int32(Var instance, int32 index, ScriptContext\* scriptContext)

{

#if FLOATVAR

return OP\_GetElementI(instance, Js::JavascriptNumber::ToVar(index, scriptContext), scriptContext);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_GetMethodElement(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer), scriptContext);

#endif

}

Var JavascriptOperators::OP\_GetMethodElement(Var instance, Var index, ScriptContext\* scriptContext)

{

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined, GetPropertyDisplayNameForError(index, scriptContext));

}

ThreadContext\* threadContext = scriptContext->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

uint32 indexVal;

PropertyRecord const \* propertyRecord;

Var value = NULL;

BOOL hasProperty = FALSE;

bool createIfNotFound = !IsJsNativeObject(object) ||

(DynamicType::Is(object->GetTypeId()) && static\_cast<DynamicObject\*>(object)->GetTypeHandler()->IsStringTypeHandler()) || JavascriptProxy::Is(object);

if (GetIndexType(index, scriptContext, &indexVal, &propertyRecord, createIfNotFound) == IndexType\_Number)

{

hasProperty = JavascriptOperators::GetItemReference(instance, object, indexVal, &value, scriptContext);

}

else

{

if (propertyRecord != nullptr)

{

hasProperty = JavascriptOperators::GetPropertyReference(instance, object, propertyRecord->GetPropertyId(), &value, scriptContext, NULL);

}

#if DBG

else

{

Assert(IsJsNativeObject(object));

JavascriptString\* indexStr = JavascriptConversion::ToString(index, scriptContext);

PropertyRecord const \* debugPropertyRecord;

scriptContext->GetOrAddPropertyRecord(indexStr->GetString(), indexStr->GetLength(), &debugPropertyRecord);

AssertMsg(!JavascriptOperators::GetPropertyReference(instance, object, debugPropertyRecord->GetPropertyId(), &value, scriptContext, NULL),

"how did this property come? See OS Bug 2727708 if you see this come from the web");

}

#endif

}

if (!hasProperty)

{

JavascriptString\* varName = JavascriptConversion::ToString(index, scriptContext);

// ES5 11.2.3 #2: We evaluate the call target but don't throw yet if target member is missing. We need to evaluate argList

// first (#3). Postpone throwing error to invoke time.

value = ThrowErrorObject::CreateThrowTypeErrorObject(scriptContext, VBSERR\_OLENoPropOrMethod, varName);

}

else if(!JavascriptConversion::IsCallable(value))

{

// ES5 11.2.3 #2: We evaluate the call target but don't throw yet if target member is missing. We need to evaluate argList

// first (#3). Postpone throwing error to invoke time.

JavascriptString\* varName = JavascriptConversion::ToString(index, scriptContext);

value = ThrowErrorObject::CreateThrowTypeErrorObject(scriptContext, JSERR\_Property\_NeedFunction, varName);

}

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

return value;

}

BOOL JavascriptOperators::OP\_SetElementI\_UInt32(Var instance, uint32 index, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

#if FLOATVAR

return OP\_SetElementI\_JIT(instance, Js::JavascriptNumber::ToVar(index, scriptContext), value, scriptContext, flags);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_SetElementI\_JIT(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer), value, scriptContext, flags);

#endif

}

BOOL JavascriptOperators::OP\_SetElementI\_Int32(Var instance, int32 index, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

#if FLOATVAR

return OP\_SetElementI\_JIT(instance, Js::JavascriptNumber::ToVar(index, scriptContext), value, scriptContext, flags);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_SetElementI\_JIT(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer), value, scriptContext, flags);

#endif

}

BOOL JavascriptOperators::OP\_SetElementI\_JIT(Var instance, Var index, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

if (TaggedNumber::Is(instance))

{

return OP\_SetElementI(instance, index, value, scriptContext, flags);

}

INT\_PTR vt = VirtualTableInfoBase::GetVirtualTable(instance);

OP\_SetElementI(instance, index, value, scriptContext, flags);

return vt != VirtualTableInfoBase::GetVirtualTable(instance);

}

BOOL JavascriptOperators::OP\_SetElementI(Var instance, Var index, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

TypeId instanceType = JavascriptOperators::GetTypeId(instance);

bool isTypedArray = (instanceType >= TypeIds\_Int8Array && instanceType <= TypeIds\_Float64Array);

if (isTypedArray)

{

if (TaggedInt::Is(index) || JavascriptNumber::Is\_NoTaggedIntCheck(index) || JavascriptString::Is(index))

{

BOOL returnValue = FALSE;

bool isNumericIndex = false;

switch (instanceType)

{

case TypeIds\_Int8Array:

{

// The typed array will deal with all possible values for the index

if (VirtualTableInfo<Int8VirtualArray>::HasVirtualTable(instance))

{

Int8VirtualArray\* int8Array = Int8VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int8Array))

{

returnValue = int8Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

else

{

Int8Array\* int8Array = Int8Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int8Array))

{

returnValue = int8Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

break;

}

case TypeIds\_Uint8Array:

{

// The typed array will deal with all possible values for the index

if (VirtualTableInfo<Uint8VirtualArray>::HasVirtualTable(instance))

{

Uint8VirtualArray\* uint8Array = Uint8VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint8Array))

{

returnValue = uint8Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

else

{

Uint8Array\* uint8Array = Uint8Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint8Array))

{

returnValue = uint8Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

break;

}

case TypeIds\_Uint8ClampedArray:

{

// The typed array will deal with all possible values for the index

if (VirtualTableInfo<Uint8ClampedVirtualArray>::HasVirtualTable(instance))

{

Uint8ClampedVirtualArray\* uint8ClampedArray = Uint8ClampedVirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint8ClampedArray))

{

returnValue = uint8ClampedArray->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

else

{

Uint8ClampedArray\* uint8ClampedArray = Uint8ClampedArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint8ClampedArray))

{

returnValue = uint8ClampedArray->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

break;

}

case TypeIds\_Int16Array:

{

// The type array will deal with all possible values for the index

if (VirtualTableInfo<Int16VirtualArray>::HasVirtualTable(instance))

{

Int16VirtualArray\* int16Array = Int16VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int16Array))

{

returnValue = int16Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

else

{

Int16Array\* int16Array = Int16Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int16Array))

{

returnValue = int16Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

break;

}

case TypeIds\_Uint16Array:

{

// The type array will deal with all possible values for the index

if (VirtualTableInfo<Uint16VirtualArray>::HasVirtualTable(instance))

{

Uint16VirtualArray\* uint16Array = Uint16VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint16Array))

{

returnValue = uint16Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

else

{

Uint16Array\* uint16Array = Uint16Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint16Array))

{

returnValue = uint16Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

break;

}

case TypeIds\_Int32Array:

{

// The type array will deal with all possible values for the index

if (VirtualTableInfo<Int32VirtualArray>::HasVirtualTable(instance))

{

Int32VirtualArray\* int32Array = Int32VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int32Array))

{

returnValue = int32Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

else

{

Int32Array\* int32Array = Int32Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int32Array))

{

returnValue = int32Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

break;

}

case TypeIds\_Uint32Array:

{

// The type array will deal with all possible values for the index

if (VirtualTableInfo<Uint32VirtualArray>::HasVirtualTable(instance))

{

Uint32VirtualArray\* uint32Array = Uint32VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint32Array))

{

returnValue = uint32Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

else

{

Uint32Array\* uint32Array = Uint32Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint32Array))

{

returnValue = uint32Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

break;

}

case TypeIds\_Float32Array:

{

// The type array will deal with all possible values for the index

if (VirtualTableInfo<Float32VirtualArray>::HasVirtualTable(instance))

{

Float32VirtualArray\* float32Array = Float32VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(float32Array))

{

returnValue = float32Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

else

{

Float32Array\* float32Array = Float32Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(float32Array))

{

returnValue = float32Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

break;

}

case TypeIds\_Float64Array:

{

// The type array will deal with all possible values for the index

if (VirtualTableInfo<Float64VirtualArray>::HasVirtualTable(instance))

{

Float64VirtualArray\* float64Array = Float64VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(float64Array))

{

returnValue = float64Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

else

{

Float64Array\* float64Array = Float64Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(float64Array))

{

returnValue = float64Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

break;

}

}

// if this was numeric index, return operation status else

// Return the result of calling the default ordinary object [[Set]] internal method (9.1.8) on O passing P, V, and Receiver as arguments.

if (isNumericIndex)

return returnValue;

}

}

else

{

if (TaggedInt::Is(index))

{

TaggedIntIndex:

switch (instanceType)

{

case TypeIds\_NativeIntArray:

case TypeIds\_NativeFloatArray:

case TypeIds\_Array: // fast path for array

{

int indexInt = TaggedInt::ToInt32(index);

if (indexInt >= 0 && scriptContext->optimizationOverrides.IsEnabledArraySetElementFastPath())

{

JavascriptArray::FromVar(instance)->SetItem((uint32)indexInt, value, flags);

return true;

}

break;

}

}

}

else if (JavascriptNumber::Is\_NoTaggedIntCheck(index))

{

double dIndexValue = JavascriptNumber::GetValue(index);

uint32 uint32Index = JavascriptConversion::ToUInt32(index, scriptContext);

if ((double)uint32Index == dIndexValue && !TaggedInt::IsOverflow(uint32Index))

{

index = TaggedInt::ToVarUnchecked(uint32Index);

goto TaggedIntIndex;

}

}

}

RecyclableObject\* object;

BOOL isNullOrUndefined = !GetPropertyObject(instance, scriptContext, &object);

Assert(object == instance || TaggedNumber::Is(instance));

if (isNullOrUndefined)

{

if (!scriptContext->GetThreadContext()->RecordImplicitException())

{

return FALSE;

}

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotSet\_NullOrUndefined, GetPropertyDisplayNameForError(index, scriptContext));

}

return JavascriptOperators::SetElementIHelper(instance, object, index, value, scriptContext, flags);

}

BOOL JavascriptOperators::SetElementIHelper(Var receiver, RecyclableObject\* object, Var index, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

PropertyString \* propertyString = nullptr;

Js::IndexType indexType;

uint32 indexVal = 0;

PropertyRecord const \* propertyRecord = nullptr;

JavascriptString \* propertyNameString = nullptr;

if (TaggedNumber::Is(receiver))

{

indexType = GetIndexType(index, scriptContext, &indexVal, &propertyRecord, true);

if (indexType == IndexType\_Number)

{

return JavascriptOperators::SetItemOnTaggedNumber(receiver, object, indexVal, value, scriptContext, flags);

}

else

{

return JavascriptOperators::SetPropertyOnTaggedNumber(receiver, object, propertyRecord->GetPropertyId(), value, scriptContext, flags);

}

}

// fastpath for PropertyStrings only if receiver == object

if (!TaggedInt::Is(index) && JavascriptString::Is(index) &&

VirtualTableInfo<Js::PropertyString>::HasVirtualTable(JavascriptString::FromVar(index)))

{

propertyString = (PropertyString \*)JavascriptString::FromVar(index);

Assert(propertyString->GetScriptContext() == scriptContext);

PropertyCache const \* cache = propertyString->GetPropertyCache();

if (receiver == object && object->GetType() == cache->type && cache->isStoreFieldEnabled)

{

#if DBG

propertyRecord = propertyString->GetPropertyRecord();

#endif

#if DBG\_DUMP

scriptContext->forinCache++;

#endif

Assert(object->GetScriptContext() == scriptContext);

Assert(!CrossSite::NeedMarshalVar(value, scriptContext));

if (cache->isInlineSlot)

{

DynamicObject::FromVar(object)->SetInlineSlot(SetSlotArguments(propertyRecord->GetPropertyId(), cache->dataSlotIndex, value));

}

else

{

DynamicObject::FromVar(object)->SetAuxSlot(SetSlotArguments(propertyRecord->GetPropertyId(), cache->dataSlotIndex, value));

}

return true;

}

propertyRecord = propertyString->GetPropertyRecord();

if (propertyRecord->IsNumeric())

{

indexType = IndexType\_Number;

indexVal = propertyRecord->GetNumericValue();

}

else

{

indexType = IndexType\_PropertyId;

}

#if DBG\_DUMP

scriptContext->forinNoCache++;

#endif

}

else

{

#if DBG\_DUMP

scriptContext->forinNoCache += (!TaggedInt::Is(index) && JavascriptString::Is(index));

#endif

indexType = GetIndexType(index, scriptContext, &indexVal, &propertyRecord, &propertyNameString, false, true);

if (scriptContext->GetThreadContext()->IsDisableImplicitCall() &&

scriptContext->GetThreadContext()->GetImplicitCallFlags() != ImplicitCall\_None)

{

// We hit an implicit call trying to convert the index, and implicit calls are disabled, so

// quit before we try to store the element.

return FALSE;

}

}

if (indexType == IndexType\_Number)

{

return JavascriptOperators::SetItem(receiver, object, indexVal, value, scriptContext, flags);

}

else if (indexType == IndexType\_JavascriptString)

{

Assert(propertyNameString);

JsUtil::CharacterBuffer<WCHAR> propertyName(propertyNameString->GetString(), propertyNameString->GetLength());

if (BuiltInPropertyRecords::NaN.Equals(propertyName))

{

// Follow SetProperty convention for NaN

return JavascriptOperators::SetProperty(receiver, object, PropertyIds::NaN, value, scriptContext, flags);

}

else if (BuiltInPropertyRecords::Infinity.Equals(propertyName))

{

// Follow SetProperty convention for Infinity

return JavascriptOperators::SetProperty(receiver, object, PropertyIds::Infinity, value, scriptContext, flags);

}

return JavascriptOperators::SetPropertyWPCache(receiver, object, propertyNameString, value, scriptContext, nullptr, flags);

}

else if (indexType == IndexType\_PropertyId)

{

Assert(propertyRecord);

PropertyId propId = propertyRecord->GetPropertyId();

if (propId == PropertyIds::NaN || propId == PropertyIds::Infinity)

{

// As we no longer convert o[x] into o.x for NaN and Infinity, we need to follow SetProperty convention for these,

// which would check for read-only properties, strict mode, etc.

// Note that "-Infinity" does not qualify as property name, so we don't have to take care of it.

return JavascriptOperators::SetProperty(receiver, object, propId, value, scriptContext, flags);

}

}

return JavascriptOperators::SetPropertyWPCache(receiver, object, propertyRecord->GetPropertyId(), value, scriptContext, propertyString, flags);

}

BOOL JavascriptOperators::OP\_SetNativeIntElementI(

Var instance,

Var aElementIndex,

int32 iValue,

ScriptContext\* scriptContext,

PropertyOperationFlags flags)

{

if (TaggedInt::Is(aElementIndex))

{

int32 indexInt = TaggedInt::ToInt32(aElementIndex);

if (indexInt >= 0 && scriptContext->optimizationOverrides.IsEnabledArraySetElementFastPath())

{

JavascriptNativeIntArray \*arr = JavascriptNativeIntArray::FromVar(instance);

if (!(arr->TryGrowHeadSegmentAndSetItem<int32, JavascriptNativeIntArray>((uint32)indexInt, iValue)))

{

arr->SetItem(indexInt, iValue);

}

return TRUE;

}

}

return JavascriptOperators::OP\_SetElementI(instance, aElementIndex, JavascriptNumber::ToVar(iValue, scriptContext), scriptContext, flags);

}

BOOL JavascriptOperators::OP\_SetNativeIntElementI\_UInt32(

Var instance,

uint32 aElementIndex,

int32 iValue,

ScriptContext\* scriptContext,

PropertyOperationFlags flags)

{

#if FLOATVAR

return OP\_SetNativeIntElementI(instance, Js::JavascriptNumber::ToVar(aElementIndex, scriptContext), iValue, scriptContext, flags);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_SetNativeIntElementI(instance, Js::JavascriptNumber::ToVarInPlace(aElementIndex, scriptContext,

(Js::JavascriptNumber \*)buffer), iValue, scriptContext, flags);

#endif

}

BOOL JavascriptOperators::OP\_SetNativeIntElementI\_Int32(

Var instance,

int aElementIndex,

int32 iValue,

ScriptContext\* scriptContext,

PropertyOperationFlags flags)

{

#if FLOATVAR

return OP\_SetNativeIntElementI(instance, Js::JavascriptNumber::ToVar(aElementIndex, scriptContext), iValue, scriptContext, flags);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_SetNativeIntElementI(instance, Js::JavascriptNumber::ToVarInPlace(aElementIndex, scriptContext,

(Js::JavascriptNumber \*)buffer), iValue, scriptContext, flags);

#endif

}

BOOL JavascriptOperators::OP\_SetNativeFloatElementI(

Var instance,

Var aElementIndex,

ScriptContext\* scriptContext,

PropertyOperationFlags flags,

double dValue)

{

if (TaggedInt::Is(aElementIndex))

{

int32 indexInt = TaggedInt::ToInt32(aElementIndex);

if (indexInt >= 0 && scriptContext->optimizationOverrides.IsEnabledArraySetElementFastPath())

{

JavascriptNativeFloatArray \*arr = JavascriptNativeFloatArray::FromVar(instance);

if (!(arr->TryGrowHeadSegmentAndSetItem<double, JavascriptNativeFloatArray>((uint32)indexInt, dValue)))

{

arr->SetItem(indexInt, dValue);

}

return TRUE;

}

}

return JavascriptOperators::OP\_SetElementI(instance, aElementIndex, JavascriptNumber::ToVarWithCheck(dValue, scriptContext), scriptContext, flags);

}

BOOL JavascriptOperators::OP\_SetNativeFloatElementI\_UInt32(

Var instance, uint32

aElementIndex,

ScriptContext\* scriptContext,

PropertyOperationFlags flags,

double dValue)

{

#if FLOATVAR

return OP\_SetNativeFloatElementI(instance, JavascriptNumber::ToVar(aElementIndex, scriptContext), scriptContext, flags, dValue);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_SetNativeFloatElementI(instance, JavascriptNumber::ToVarInPlace(aElementIndex, scriptContext,

(Js::JavascriptNumber \*)buffer), scriptContext, flags, dValue);

#endif

}

BOOL JavascriptOperators::OP\_SetNativeFloatElementI\_Int32(

Var instance,

int aElementIndex,

ScriptContext\* scriptContext,

PropertyOperationFlags flags,

double dValue)

{

#if FLOATVAR

return OP\_SetNativeFloatElementI(instance, JavascriptNumber::ToVar(aElementIndex, scriptContext), scriptContext, flags, dValue);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_SetNativeFloatElementI(instance, JavascriptNumber::ToVarInPlace(aElementIndex, scriptContext,

(Js::JavascriptNumber \*)buffer), scriptContext, flags, dValue);

#endif

}

BOOL JavascriptOperators::OP\_Memcopy(Var dstInstance, int32 dstStart, Var srcInstance, int32 srcStart, int32 length, ScriptContext\* scriptContext)

{

if (length <= 0)

{

return true;

}

TypeId instanceType = JavascriptOperators::GetTypeId(srcInstance);

if (instanceType != JavascriptOperators::GetTypeId(dstInstance))

{

return false;

}

if (srcStart != dstStart)

{

return false;

}

BOOL returnValue = false;

switch (instanceType)

{

case TypeIds\_Int8Array:

{

// The typed array will deal with all possible values for the index

returnValue = Int8Array::FromVar(dstInstance)->DirectSetItemAtRange(Int8Array::FromVar(srcInstance), srcStart, dstStart, length, JavascriptConversion::ToInt8);

break;

}

case TypeIds\_Uint8Array:

{

returnValue = Uint8Array::FromVar(dstInstance)->DirectSetItemAtRange(Uint8Array::FromVar(srcInstance), srcStart, dstStart, length, JavascriptConversion::ToUInt8);

break;

}

case TypeIds\_Uint8ClampedArray:

{

returnValue = Uint8ClampedArray::FromVar(dstInstance)->DirectSetItemAtRange(Uint8ClampedArray::FromVar(srcInstance), srcStart, dstStart, length, JavascriptConversion::ToUInt8Clamped);

break;

}

case TypeIds\_Int16Array:

{

returnValue = Int16Array::FromVar(dstInstance)->DirectSetItemAtRange(Int16Array::FromVar(srcInstance), srcStart, dstStart, length, JavascriptConversion::ToInt16);

break;

}

case TypeIds\_Uint16Array:

{

returnValue = Uint16Array::FromVar(dstInstance)->DirectSetItemAtRange(Uint16Array::FromVar(srcInstance), srcStart, dstStart, length, JavascriptConversion::ToUInt16);

break;

}

case TypeIds\_Int32Array:

{

returnValue = Int32Array::FromVar(dstInstance)->DirectSetItemAtRange(Int32Array::FromVar(srcInstance), srcStart, dstStart, length, JavascriptConversion::ToInt32);

break;

}

case TypeIds\_Uint32Array:

{

returnValue = Uint32Array::FromVar(dstInstance)->DirectSetItemAtRange(Uint32Array::FromVar(srcInstance), srcStart, dstStart, length, JavascriptConversion::ToUInt32);

break;

}

case TypeIds\_Array:

case TypeIds\_NativeIntArray:

{

if (dstStart < 0 || srcStart < 0)

{

// This is not supported, Bailout

break;

}

// Upper bounds check for source array

uint32 end;

if (UInt32Math::Add(srcStart, length, &end) || end > ((ArrayObject\*)srcInstance)->GetLength())

{

return false;

}

if (scriptContext->optimizationOverrides.IsEnabledArraySetElementFastPath())

{

INT\_PTR vt = VirtualTableInfoBase::GetVirtualTable(dstInstance);

if (instanceType == TypeIds\_Array)

{

returnValue = JavascriptArray::FromVar(dstInstance)->DirectSetItemAtRangeFromArray<Var>(dstStart, length, JavascriptArray::FromVar(srcInstance), srcStart);

}

else

{

returnValue = JavascriptArray::FromVar(dstInstance)->DirectSetItemAtRangeFromArray<int32>(dstStart, length, JavascriptArray::FromVar(srcInstance), srcStart);

}

returnValue &= vt == VirtualTableInfoBase::GetVirtualTable(dstInstance);

}

break;

}

default:

{

AssertMsg(false, "We don't support this type for memcopy yet.");

break;

}

}

return returnValue;

}

BOOL JavascriptOperators::OP\_Memset(Var instance, int32 start, Var value, int32 length, ScriptContext\* scriptContext)

{

if (length <= 0)

{

return true;

}

TypeId instanceType = JavascriptOperators::GetTypeId(instance);

BOOL returnValue = false;

// The typed array will deal with all possible values for the index

#define MEMSET\_TYPED\_ARRAY(type, conversion) type ## ::FromVar(instance)->DirectSetItemAtRange(start, length, value, JavascriptConversion:: ## conversion)

switch (instanceType)

{

case TypeIds\_Int8Array:

{

returnValue = MEMSET\_TYPED\_ARRAY(Int8Array, ToInt8);

break;

}

case TypeIds\_Uint8Array:

{

returnValue = MEMSET\_TYPED\_ARRAY(Uint8Array, ToUInt8);

break;

}

case TypeIds\_Uint8ClampedArray:

{

returnValue = MEMSET\_TYPED\_ARRAY(Uint8ClampedArray, ToUInt8Clamped);

break;

}

case TypeIds\_Int16Array:

{

returnValue = MEMSET\_TYPED\_ARRAY(Int16Array, ToInt16);

break;

}

case TypeIds\_Uint16Array:

{

returnValue = MEMSET\_TYPED\_ARRAY(Uint16Array, ToUInt16);

break;

}

case TypeIds\_Int32Array:

{

returnValue = MEMSET\_TYPED\_ARRAY(Int32Array, ToInt32);

break;

}

case TypeIds\_Uint32Array:

{

returnValue = MEMSET\_TYPED\_ARRAY(Uint32Array, ToUInt32);

break;

}

case TypeIds\_Float32Array:

{

returnValue = MEMSET\_TYPED\_ARRAY(Float32Array, ToFloat);

break;

}

case TypeIds\_Float64Array:

{

returnValue = MEMSET\_TYPED\_ARRAY(Float64Array, ToNumber);

break;

}

case TypeIds\_NativeFloatArray:

case TypeIds\_NativeIntArray:

case TypeIds\_Array:

{

if (start < 0)

{

for (start; start < 0 && length > 0; ++start, --length)

{

if (!OP\_SetElementI(instance, JavascriptNumber::ToVar(start, scriptContext), value, scriptContext))

{

return false;

}

}

}

if (scriptContext->optimizationOverrides.IsEnabledArraySetElementFastPath())

{

INT\_PTR vt = VirtualTableInfoBase::GetVirtualTable(instance);

if (instanceType == TypeIds\_Array)

{

returnValue = JavascriptArray::FromVar(instance)->DirectSetItemAtRange<Var>(start, length, value);

}

else if (instanceType == TypeIds\_NativeIntArray)

{

returnValue = JavascriptArray::FromVar(instance)->DirectSetItemAtRange<int32>(start, length, JavascriptConversion::ToInt32(value, scriptContext));

}

else

{

returnValue = JavascriptArray::FromVar(instance)->DirectSetItemAtRange<double>(start, length, JavascriptConversion::ToNumber(value, scriptContext));

}

returnValue &= vt == VirtualTableInfoBase::GetVirtualTable(instance);

}

break;

}

default:

{

AssertMsg(false, "We don't support this type for memset yet.");

break;

}

}

#undef MEMSET\_TYPED\_ARRAY

return returnValue;

}

Var JavascriptOperators::OP\_DeleteElementI\_UInt32(Var instance, uint32 index, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags)

{

#if FLOATVAR

return OP\_DeleteElementI(instance, Js::JavascriptNumber::ToVar(index, scriptContext), scriptContext, propertyOperationFlags);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_DeleteElementI(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer), scriptContext, propertyOperationFlags);

#endif

}

Var JavascriptOperators::OP\_DeleteElementI\_Int32(Var instance, int32 index, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags)

{

#if FLOATVAR

return OP\_DeleteElementI(instance, Js::JavascriptNumber::ToVar(index, scriptContext), scriptContext, propertyOperationFlags);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_DeleteElementI(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer), scriptContext, propertyOperationFlags);

#endif

}

Var JavascriptOperators::OP\_DeleteElementI(Var instance, Var index, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags)

{

if(TaggedNumber::Is(instance))

{

return scriptContext->GetLibrary()->GetTrue();

}

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

TypeId typeId = JavascriptOperators::GetTypeId(instance);

if (typeId == TypeIds\_Null || typeId == TypeIds\_Undefined)

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotDelete\_NullOrUndefined, GetPropertyDisplayNameForError(index, scriptContext));

}

RecyclableObject\* object = RecyclableObject::FromVar(instance);

uint32 indexVal;

PropertyRecord const \* propertyRecord;

BOOL result = TRUE;

bool createIfNotFound = !IsJsNativeObject(object) ||

(DynamicType::Is(object->GetTypeId()) && static\_cast<DynamicObject\*>(object)->GetTypeHandler()->IsStringTypeHandler()) || JavascriptProxy::Is(object);

if (GetIndexType(index, scriptContext, &indexVal, &propertyRecord, createIfNotFound) == IndexType\_Number)

{

result = JavascriptOperators::DeleteItem(object, indexVal, propertyOperationFlags);

}

else

{

if (propertyRecord)

{

result = JavascriptOperators::DeleteProperty(object, propertyRecord->GetPropertyId(), propertyOperationFlags);

}

#if DBG

else

{

Assert(IsJsNativeObject(object));

JavascriptString\* indexStr = JavascriptConversion::ToString(index, scriptContext);

PropertyRecord const \* debugPropertyRecord;

scriptContext->GetOrAddPropertyRecord(indexStr->GetString(), indexStr->GetLength(), &debugPropertyRecord);

AssertMsg(JavascriptOperators::DeleteProperty(object, debugPropertyRecord->GetPropertyId(), propertyOperationFlags), "delete should have been true. See OS Bug 2727708 if you see this come from the web");

}

#endif

}

return scriptContext->GetLibrary()->CreateBoolean(result);

}

Var JavascriptOperators::OP\_GetLength(Var instance, ScriptContext\* scriptContext)

{

return JavascriptOperators::OP\_GetProperty(instance, PropertyIds::length, scriptContext);

}

\_\_inline Var JavascriptOperators::GetThisFromModuleRoot(Var thisVar)

{

RootObjectBase \* rootObject = static\_cast<RootObjectBase\*>(thisVar);

RecyclableObject\* hostObject = rootObject->GetHostObject();

//

// if the module root has the host object, use that as "this"

//

if (hostObject)

{

thisVar = hostObject->GetHostDispatchVar();

}

return thisVar;

}

\_\_inline void JavascriptOperators::TryLoadRoot(Var& thisVar, TypeId typeId, int moduleID, ScriptContext\* scriptContext)

{

bool loadRoot = false;

if (JavascriptOperators::IsUndefinedOrNullType(typeId) || typeId == TypeIds\_ActivationObject)

{

loadRoot = true;

}

else if (typeId == TypeIds\_HostDispatch)

{

TypeId remoteTypeId;

if (RecyclableObject::FromVar(thisVar)->GetRemoteTypeId(&remoteTypeId))

{

if (remoteTypeId == TypeIds\_Null || remoteTypeId == TypeIds\_Undefined || remoteTypeId == TypeIds\_ActivationObject)

{

loadRoot = true;

}

}

}

if (loadRoot)

{

if (moduleID == 0)

{

thisVar = JavascriptOperators::OP\_LdRoot(scriptContext)->ToThis();

}

else

{

Js::ModuleRoot \* moduleRoot = JavascriptOperators::GetModuleRoot(moduleID, scriptContext);

if (moduleRoot == nullptr)

{

Assert(false);

thisVar = scriptContext->GetLibrary()->GetUndefined();

}

else

{

thisVar = GetThisFromModuleRoot(moduleRoot);

}

}

}

}

Var JavascriptOperators::OP\_GetThis(Var thisVar, int moduleID, ScriptContext\* scriptContext)

{

//

// if "this" is null or undefined

// Pass the global object

// Else

// Pass ToObject(this)

//

TypeId typeId = JavascriptOperators::GetTypeId(thisVar);

Assert(!JavascriptOperators::IsThisSelf(typeId));

return JavascriptOperators::GetThisHelper(thisVar, typeId, moduleID, scriptContext);

}

Var JavascriptOperators::OP\_GetThisNoFastPath(Var thisVar, int moduleID, ScriptContext\* scriptContext)

{

TypeId typeId = JavascriptOperators::GetTypeId(thisVar);

if (JavascriptOperators::IsThisSelf(typeId))

{

Assert(typeId != TypeIds\_GlobalObject || ((Js::GlobalObject\*)thisVar)->ToThis() == thisVar);

Assert(typeId != TypeIds\_ModuleRoot || JavascriptOperators::GetThisFromModuleRoot(thisVar) == thisVar);

return thisVar;

}

return JavascriptOperators::GetThisHelper(thisVar, typeId, moduleID, scriptContext);

}

bool JavascriptOperators::IsThisSelf(TypeId typeId)

{

return (JavascriptOperators::IsObjectType(typeId) && ! JavascriptOperators::IsSpecialObjectType(typeId));

}

Var JavascriptOperators::GetThisHelper(Var thisVar, TypeId typeId, int moduleID, ScriptContext \*scriptContext)

{

if (! JavascriptOperators::IsObjectType(typeId) && ! JavascriptOperators::IsUndefinedOrNullType(typeId))

{

#if !FLOATVAR

// We allowed stack number to be used as the "this" for getter and setter activation of

// n.x and n[prop], where n is the Javascript Number

return JavascriptOperators::ToObject(

JavascriptNumber::BoxStackNumber(thisVar, scriptContext), scriptContext);

#else

return JavascriptOperators::ToObject(thisVar, scriptContext);

#endif

}

else

{

TryLoadRoot(thisVar, typeId, moduleID, scriptContext);

return thisVar;

}

}

Var JavascriptOperators::OP\_StrictGetThis(Var thisVar, ScriptContext\* scriptContext)

{

TypeId typeId = JavascriptOperators::GetTypeId(thisVar);

if (typeId == TypeIds\_ActivationObject)

{

return scriptContext->GetLibrary()->GetUndefined();

}

return thisVar;

}

BOOL JavascriptOperators::GetRemoteTypeId(Var aValue, TypeId\* typeId)

{

if (GetTypeId(aValue) != TypeIds\_HostDispatch)

{

return FALSE;

}

return RecyclableObject::FromVar(aValue)->GetRemoteTypeId(typeId);

}

BOOL JavascriptOperators::IsJsNativeObject(Var aValue)

{

switch(GetTypeId(aValue))

{

case TypeIds\_Object:

case TypeIds\_Function:

case TypeIds\_Array:

case TypeIds\_NativeIntArray:

#if ENABLE\_COPYONACCESS\_ARRAY

case TypeIds\_CopyOnAccessNativeIntArray:

#endif

case TypeIds\_NativeFloatArray:

case TypeIds\_ES5Array:

case TypeIds\_Date:

case TypeIds\_WinRTDate:

case TypeIds\_RegEx:

case TypeIds\_Error:

case TypeIds\_BooleanObject:

case TypeIds\_NumberObject:

case TypeIds\_StringObject:

case TypeIds\_Symbol:

case TypeIds\_SymbolObject:

//case TypeIds\_GlobalObject:

//case TypeIds\_ModuleRoot:

//case TypeIds\_HostObject:

case TypeIds\_Arguments:

case TypeIds\_ActivationObject:

case TypeIds\_Map:

case TypeIds\_Set:

case TypeIds\_WeakMap:

case TypeIds\_WeakSet:

case TypeIds\_ArrayIterator:

case TypeIds\_MapIterator:

case TypeIds\_SetIterator:

case TypeIds\_StringIterator:

case TypeIds\_Generator:

case TypeIds\_Promise:

case TypeIds\_Proxy:

return true;

default:

return false;

}

}

RecyclableObject\* JavascriptOperators::GetPrototype(RecyclableObject\* instance)

{

if (JavascriptOperators::GetTypeId(instance) == TypeIds\_Null)

{

return instance;

}

return instance->GetPrototype();

}

RecyclableObject\* JavascriptOperators::OP\_GetPrototype(Var instance, ScriptContext\* scriptContext)

{

if (TaggedNumber::Is(instance))

{

return scriptContext->GetLibrary()->GetNumberPrototype();

}

else if (JavascriptOperators::GetTypeId(instance) != TypeIds\_Null)

{

return JavascriptOperators::GetPrototype(RecyclableObject::FromVar(instance));

}

else

{

return scriptContext->GetLibrary()->GetNull();

}

}

BOOL JavascriptOperators::OP\_BrFncEqApply(Var instance, ScriptContext \*scriptContext)

{

// JavascriptFunction && !HostDispatch

if (JavascriptOperators::GetTypeId(instance) == TypeIds\_Function)

{

FunctionProxy \*bod= ((JavascriptFunction\*)instance)->GetFunctionProxy();

if (bod != nullptr)

{

return bod->GetDirectEntryPoint(bod->GetDefaultEntryPointInfo()) == &Js::JavascriptFunction::EntryApply;

}

else

{

FunctionInfo\* info = ((JavascriptFunction \*)instance)->GetFunctionInfo();

if (info != nullptr)

{

return &Js::JavascriptFunction::EntryApply == info->GetOriginalEntryPoint();

}

else

{

return false;

}

}

}

return false;

}

BOOL JavascriptOperators::OP\_BrFncNeqApply(Var instance, ScriptContext \*scriptContext)

{

// JavascriptFunction and !HostDispatch

if (JavascriptOperators::GetTypeId(instance) == TypeIds\_Function)

{

FunctionProxy \*bod = ((JavascriptFunction \*)instance)->GetFunctionProxy();

if (bod != nullptr)

{

return bod->GetDirectEntryPoint(bod->GetDefaultEntryPointInfo()) != &Js::JavascriptFunction::EntryApply;

}

else

{

FunctionInfo\* info = ((JavascriptFunction \*)instance)->GetFunctionInfo();

if (info != nullptr)

{

return &Js::JavascriptFunction::EntryApply != info->GetOriginalEntryPoint();

}

else

{

return true;

}

}

}

return true;

}

BOOL JavascriptOperators::OP\_BrHasSideEffects(int se, ScriptContext\* scriptContext)

{

return (scriptContext->optimizationOverrides.GetSideEffects() & se) != SideEffects\_None;

}

BOOL JavascriptOperators::OP\_BrNotHasSideEffects(int se, ScriptContext\* scriptContext)

{

return (scriptContext->optimizationOverrides.GetSideEffects() & se) == SideEffects\_None;

}

// returns NULL if there is no more elements to enumerate.

Var JavascriptOperators::OP\_BrOnEmpty(ForInObjectEnumerator \* aEnumerator)

{

PropertyId id;

return aEnumerator->GetCurrentAndMoveNext(id);

}

ForInObjectEnumerator \* JavascriptOperators::OP\_GetForInEnumerator(Var enumerable, ScriptContext\* scriptContext)

{

RecyclableObject\* enumerableObject;

bool isCrossSite;

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(enumerable);

#endif

if (GetPropertyObject(enumerable, scriptContext, &enumerableObject))

{

isCrossSite = enumerableObject->GetScriptContext() != scriptContext;

}

else

{

enumerableObject = nullptr;

isCrossSite = false;

}

if (!isCrossSite)

{

ForInObjectEnumerator \* enumerator = scriptContext->GetLibrary()->GetAndClearForInEnumeratorCache();

if(enumerator != NULL)

{

enumerator->Initialize(enumerableObject, scriptContext);

return enumerator;

}

}

return RecyclerNew(scriptContext->GetRecycler(), ForInObjectEnumerator, enumerableObject, scriptContext);

}

void JavascriptOperators::OP\_ReleaseForInEnumerator(ForInObjectEnumerator \* enumerator, ScriptContext\* scriptContext)

{

// Debugger SetNextStatement may skip OP\_GetForInEnumerator and result in NULL ForInObjectEnumerator here. See Win8 391556

if (enumerator && enumerator->CanBeReused())

{

enumerator->Clear();

scriptContext->GetLibrary()->SetForInEnumeratorCache(enumerator);

}

}

Js::Var JavascriptOperators::OP\_CmEq\_A(Var a, Var b, ScriptContext\* scriptContext)

{

return JavascriptBoolean::ToVar(JavascriptOperators::Equal(a, b, scriptContext), scriptContext);

}

Var JavascriptOperators::OP\_CmNeq\_A(Var a, Var b, ScriptContext\* scriptContext)

{

return JavascriptBoolean::ToVar(JavascriptOperators::NotEqual(a,b,scriptContext), scriptContext);

}

Var JavascriptOperators::OP\_CmSrEq\_A(Var a, Var b, ScriptContext\* scriptContext)

{

return JavascriptBoolean::ToVar(JavascriptOperators::StrictEqual(a, b, scriptContext), scriptContext);

}

Var JavascriptOperators::OP\_CmSrEq\_String(Var a, Var b, ScriptContext \*scriptContext)

{

return JavascriptBoolean::ToVar(JavascriptOperators::StrictEqualString(a, b), scriptContext);

}

Var JavascriptOperators::OP\_CmSrEq\_EmptyString(Var a, ScriptContext \*scriptContext)

{

return JavascriptBoolean::ToVar(JavascriptOperators::StrictEqualEmptyString(a), scriptContext);

}

Var JavascriptOperators::OP\_CmSrNeq\_A(Var a, Var b, ScriptContext\* scriptContext)

{

return JavascriptBoolean::ToVar(JavascriptOperators::NotStrictEqual(a, b, scriptContext), scriptContext);

}

Var JavascriptOperators::OP\_CmLt\_A(Var a, Var b, ScriptContext\* scriptContext)

{

return JavascriptBoolean::ToVar(JavascriptOperators::Less(a, b, scriptContext), scriptContext);

}

Var JavascriptOperators::OP\_CmLe\_A(Var a, Var b, ScriptContext\* scriptContext)

{

return JavascriptBoolean::ToVar(JavascriptOperators::LessEqual(a, b, scriptContext), scriptContext);

}

Var JavascriptOperators::OP\_CmGt\_A(Var a, Var b, ScriptContext\* scriptContext)

{

return JavascriptBoolean::ToVar(JavascriptOperators::Greater(a, b, scriptContext), scriptContext);

}

Var JavascriptOperators::OP\_CmGe\_A(Var a, Var b, ScriptContext\* scriptContext)

{

return JavascriptBoolean::ToVar(JavascriptOperators::GreaterEqual(a, b, scriptContext), scriptContext);

}

DetachedStateBase\* JavascriptOperators::DetachVarAndGetState(Var var)

{

switch (GetTypeId(var))

{

case TypeIds\_ArrayBuffer:

return Js::ArrayBuffer::FromVar(var)->DetachAndGetState();

default:

if (!Js::RecyclableObject::FromVar(var)->IsExternal())

{

AssertMsg(false, "We should explicitly have a case statement for each non-external object that can be detached.");

}

return nullptr;

}

}

bool JavascriptOperators::IsObjectDetached(Var var)

{

switch (GetTypeId(var))

{

case TypeIds\_ArrayBuffer:

return Js::ArrayBuffer::FromVar(var)->IsDetached();

default:

return false;

}

}

Var JavascriptOperators::NewVarFromDetachedState(DetachedStateBase\* state, JavascriptLibrary \*library)

{

switch (state->GetTypeId())

{

case TypeIds\_ArrayBuffer:

return Js::ArrayBuffer::NewFromDetachedState(state, library);

break;

default:

AssertMsg(false, "We should explicitly have a case statement for each object which has detached state.");

return nullptr;

}

}

DynamicType \*

JavascriptOperators::EnsureObjectLiteralType(ScriptContext\* scriptContext, const Js::PropertyIdArray \*propIds, DynamicType \*\* literalType)

{

DynamicType \* newType = \*literalType;

if (newType != nullptr)

{

if (!newType->GetIsShared())

{

newType->ShareType();

}

}

else

{

DynamicType\* objectType =

FunctionBody::DoObjectHeaderInliningForObjectLiteral(propIds, scriptContext)

? scriptContext->GetLibrary()->GetObjectHeaderInlinedLiteralType((uint16)propIds->count)

: scriptContext->GetLibrary()->GetObjectLiteralType(

static\_cast<PropertyIndex>(

min(propIds->count, static\_cast<uint32>(MaxPreInitializedObjectTypeInlineSlotCount))));

newType = PathTypeHandlerBase::CreateTypeForNewScObject(scriptContext, objectType, propIds, false);

\*literalType = newType;

}

Assert(GetLiteralInlineSlotCapacity(propIds, scriptContext) == newType->GetTypeHandler()->GetInlineSlotCapacity());

Assert(newType->GetTypeHandler()->GetSlotCapacity() >= 0);

Assert(GetLiteralSlotCapacity(propIds, scriptContext) == (uint)newType->GetTypeHandler()->GetSlotCapacity());

return newType;

}

Var JavascriptOperators::NewScObjectLiteral(ScriptContext\* scriptContext, const Js::PropertyIdArray \*propIds, DynamicType \*\* literalType)

{

Assert(propIds->count != 0);

Assert(!propIds->hadDuplicates); // duplicates are removed by parser

#ifdef PROFILE\_OBJECT\_LITERALS

// Empty objects not counted in the object literal counts

scriptContext->objectLiteralInstanceCount++;

if (propIds->count > scriptContext->objectLiteralMaxLength)

{

scriptContext->objectLiteralMaxLength = propIds->count;

}

#endif

DynamicType\* newType = EnsureObjectLiteralType(scriptContext, propIds, literalType);

DynamicObject\* instance = DynamicObject::New(scriptContext->GetRecycler(), newType);

if (!newType->GetIsShared())

{

newType->GetTypeHandler()->SetSingletonInstanceIfNeeded(instance);

}

#ifdef PROFILE\_OBJECT\_LITERALS

else

{

scriptContext->objectLiteralCacheCount++;

}

#endif

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(instance));

// can't auto-proxy here as object literal is not exactly "new" object and cannot be intercepted as proxy.

return instance;

}

uint JavascriptOperators::GetLiteralSlotCapacity(Js::PropertyIdArray const \* propIds, ScriptContext \*const scriptContext)

{

const uint inlineSlotCapacity = GetLiteralInlineSlotCapacity(propIds, scriptContext);

return DynamicTypeHandler::RoundUpSlotCapacity(propIds->count, static\_cast<PropertyIndex>(inlineSlotCapacity));

}

uint JavascriptOperators::GetLiteralInlineSlotCapacity(

Js::PropertyIdArray const \* propIds,

ScriptContext \*const scriptContext)

{

if (propIds->hadDuplicates)

{

return 0;

}

return

FunctionBody::DoObjectHeaderInliningForObjectLiteral(propIds, scriptContext)

? DynamicTypeHandler::RoundUpObjectHeaderInlinedInlineSlotCapacity(static\_cast<PropertyIndex>(propIds->count))

: DynamicTypeHandler::RoundUpInlineSlotCapacity(

static\_cast<PropertyIndex>(

min(propIds->count, static\_cast<uint32>(MaxPreInitializedObjectTypeInlineSlotCount))));

}

Var JavascriptOperators::OP\_InitCachedScope(Var varFunc, const Js::PropertyIdArray \*propIds, DynamicType \*\* literalType, bool formalsAreLetDecls, ScriptContext \*scriptContext)

{

ScriptFunction \*func = JavascriptGeneratorFunction::Is(varFunc) ?

JavascriptGeneratorFunction::FromVar(varFunc)->GetGeneratorVirtualScriptFunction() :

ScriptFunction::FromVar(varFunc);

#ifdef PROFILE\_OBJECT\_LITERALS

// Empty objects not counted in the object literal counts

scriptContext->objectLiteralInstanceCount++;

if (propIds->count > scriptContext->objectLiteralMaxLength)

{

scriptContext->objectLiteralMaxLength = propIds->count;

}

#endif

PropertyId cachedFuncCount = ActivationObjectEx::GetCachedFuncCount(propIds);

PropertyId firstFuncSlot = ActivationObjectEx::GetFirstFuncSlot(propIds);

PropertyId firstVarSlot = ActivationObjectEx::GetFirstVarSlot(propIds);

PropertyId lastFuncSlot = Constants::NoProperty;

if (firstFuncSlot != Constants::NoProperty)

{

if (firstVarSlot == Constants::NoProperty)

{

lastFuncSlot = propIds->count - 1;

}

else

{

lastFuncSlot = firstVarSlot - 1;

}

}

DynamicType \*type = \*literalType;

if (type != nullptr)

{

#ifdef PROFILE\_OBJECT\_LITERALS

scriptContext->objectLiteralCacheCount++;

#endif

}

else

{

type = scriptContext->GetLibrary()->GetActivationObjectType();

if (formalsAreLetDecls)

{

uint formalsSlotLimit = (firstFuncSlot != Constants::NoProperty) ? (uint)firstFuncSlot :

(firstVarSlot != Constants::NoProperty) ? (uint)firstVarSlot :

propIds->count;

type = PathTypeHandlerBase::CreateNewScopeObject(scriptContext, type, propIds, PropertyLet, formalsSlotLimit);

}

else

{

type = PathTypeHandlerBase::CreateNewScopeObject(scriptContext, type, propIds);

}

\*literalType = type;

}

Var undef = scriptContext->GetLibrary()->GetUndefined();

ActivationObjectEx \*scopeObjEx = func->GetCachedScope();

if (scopeObjEx && scopeObjEx->IsCommitted())

{

scopeObjEx->ReplaceType(type);

scopeObjEx->SetCommit(false);

#if DBG

for (uint i = firstVarSlot; i < propIds->count; i++)

{

AssertMsg(scopeObjEx->GetSlot(i) == undef, "Var attached to cached scope");

}

#endif

}

else

{

ActivationObjectEx \*tmp = RecyclerNewPlus(scriptContext->GetRecycler(), (cachedFuncCount == 0 ? 0 : cachedFuncCount - 1) \* sizeof(FuncCacheEntry), ActivationObjectEx, type, func, cachedFuncCount, firstFuncSlot, lastFuncSlot);

if (!scopeObjEx)

{

func->SetCachedScope(tmp);

}

scopeObjEx = tmp;

for (uint i = firstVarSlot; i < propIds->count; i++)

{

scopeObjEx->SetSlot(SetSlotArguments(propIds->elements[i], i, undef));

}

}

return scopeObjEx;

}

void JavascriptOperators::OP\_InvalidateCachedScope(void\* varEnv, int32 envIndex)

{

FrameDisplay \*disp = (FrameDisplay\*)varEnv;

RecyclableObject \*objScope = RecyclableObject::FromVar(disp->GetItem(envIndex));

objScope->InvalidateCachedScope();

}

void JavascriptOperators::OP\_InitCachedFuncs(Var varScope, FrameDisplay \*pDisplay, const FuncInfoArray \*info, ScriptContext \*scriptContext)

{

ActivationObjectEx \*scopeObj = (ActivationObjectEx\*)ActivationObjectEx::FromVar(varScope);

Assert(scopeObj->GetTypeHandler()->GetInlineSlotCapacity() == 0);

ScriptFunction \*func;

FuncCacheEntry \*entry;

FunctionProxy \*proxy;

uint scopeSlot;

uint funcCount = info->count;

if (funcCount == 0)

{

// Degenerate case: no nested funcs at all

return;

}

if (scopeObj->HasCachedFuncs())

{

for (uint i = 0; i < funcCount; i++)

{

entry = scopeObj->GetFuncCacheEntry(i);

func = entry->func;

proxy = func->GetFunctionProxy();

if (proxy != proxy->GetFunctionProxy())

{

// The FunctionProxy has changed since the object was cached, e.g., due to execution

// of a deferred function through a different object.

proxy = proxy->GetFunctionProxy();

func->SetFunctionInfo(proxy);

}

// Reset the function's type to the default type with no properties

// Use the cached type on the function proxy rather than the type in the func cache entry

// CONSIDER: Stop caching the function types in the scope object

func->ReplaceType(proxy->EnsureDeferredPrototypeType());

func->ResetConstructorCacheToDefault();

scopeSlot = info->elements[i].scopeSlot;

if (scopeSlot != Constants::NoProperty)

{

// CONSIDER: Store property IDs in FuncInfoArray in debug builds so we can properly assert in SetAuxSlot

scopeObj->SetAuxSlot(SetSlotArguments(Constants::NoProperty, scopeSlot, entry->func));

}

}

return;

}

// No cached functions, so create them and cache them.

JavascriptFunction \*funcParent = scopeObj->GetParentFunc();

for (uint i = 0; i < funcCount; i++)

{

const FuncInfoEntry \*entry = &info->elements[i];

uint nestedIndex = entry->nestedIndex;

scopeSlot = entry->scopeSlot;

proxy = funcParent->GetFunctionBody()->GetNestedFunc(nestedIndex);

func = scriptContext->GetLibrary()->CreateScriptFunction(proxy);

func->SetEnvironment(pDisplay);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_FUNCTION(func, EtwTrace::GetFunctionId(proxy)));

scopeObj->SetCachedFunc(i, func);

if (scopeSlot != Constants::NoProperty)

{

// CONSIDER: Store property IDs in FuncInfoArray in debug builds so we can properly assert in SetAuxSlot

scopeObj->SetAuxSlot(SetSlotArguments(Constants::NoProperty, scopeSlot, func));

}

}

}

Var JavascriptOperators::AddVarsToArraySegment(SparseArraySegment<Var> \* segment, const Js::VarArray \*vars)

{

uint32 count = vars->count;

Assert(segment->left == 0);

Assert(count <= segment->size);

if(count > segment->length)

{

segment->length = count;

}

js\_memcpy\_s(segment->elements, sizeof(Var) \* segment->length, vars->elements, sizeof(Var) \* count);

return segment;

}

void JavascriptOperators::AddIntsToArraySegment(SparseArraySegment<int32> \* segment, const Js::AuxArray<int32> \*ints)

{

uint32 count = ints->count;

Assert(segment->left == 0);

Assert(count <= segment->size);

if(count > segment->length)

{

segment->length = count;

}

js\_memcpy\_s(segment->elements, sizeof(int32) \* segment->length, ints->elements, sizeof(int32) \* count);

}

void JavascriptOperators::AddFloatsToArraySegment(SparseArraySegment<double> \* segment, const Js::AuxArray<double> \*doubles)

{

uint32 count = doubles->count;

Assert(segment->left == 0);

Assert(count <= segment->size);

if(count > segment->length)

{

segment->length = count;

}

js\_memcpy\_s(segment->elements, sizeof(double) \* segment->length, doubles->elements, sizeof(double) \* count);

}

RecyclableObject \* JavascriptOperators::GetPrototypeObject(RecyclableObject \* constructorFunction, ScriptContext \* scriptContext)

{

Var prototypeProperty = JavascriptOperators::GetProperty(constructorFunction, PropertyIds::prototype, scriptContext);

RecyclableObject\* prototypeObject;

PrototypeObject(prototypeProperty, constructorFunction, scriptContext, &prototypeObject);

return prototypeObject;

}

RecyclableObject \* JavascriptOperators::GetPrototypeObjectForConstructorCache(RecyclableObject \* constructor, ScriptContext\* requestContext, bool& canBeCached)

{

PropertyValueInfo info;

Var prototypeValue;

RecyclableObject\* prototypeObject;

canBeCached = false;

// Do a local property lookup. Since a function's prototype property is a non-configurable data property, we don't need to worry

// about the prototype being an accessor property, whose getter returns different values every time it's called.

if (constructor->GetProperty(constructor, PropertyIds::prototype, &prototypeValue, &info, requestContext))

{

if (!JavascriptOperators::PrototypeObject(prototypeValue, constructor, requestContext, &prototypeObject))

{

// The value returned by the property lookup is not a valid prototype object, default to object prototype.

Assert(prototypeObject == constructor->GetLibrary()->GetObjectPrototype());

}

// For these scenarios, we do not want to populate the cache.

if (constructor->GetScriptContext() != requestContext || info.GetInstance() != constructor)

{

return prototypeObject;

}

}

else

{

// It's ok to cache Object.prototype, because Object.prototype cannot be overwritten.

prototypeObject = constructor->GetLibrary()->GetObjectPrototype();

}

canBeCached = true;

return prototypeObject;

}

bool JavascriptOperators::PrototypeObject(Var prototypeProperty, RecyclableObject \* constructorFunction, ScriptContext \* scriptContext, RecyclableObject\*\* prototypeObject)

{

TypeId prototypeType = JavascriptOperators::GetTypeId(prototypeProperty);

if (JavascriptOperators::IsObjectType(prototypeType))

{

\*prototypeObject = RecyclableObject::FromVar(prototypeProperty);

return true;

}

\*prototypeObject = constructorFunction->GetLibrary()->GetObjectPrototype();

return false;

}

FunctionInfo\* JavascriptOperators::GetConstructorFunctionInfo(Var instance, ScriptContext \* scriptContext)

{

TypeId typeId = JavascriptOperators::GetTypeId(instance);

if (typeId == TypeIds\_Function)

{

JavascriptFunction \* function = JavascriptFunction::FromVar(instance);

return function->GetFunctionInfo();

}

if (typeId != TypeIds\_HostDispatch && typeId != TypeIds\_Proxy)

{

if (typeId == TypeIds\_Null)

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedObject);

}

JavascriptError::ThrowTypeError(scriptContext, VBSERR\_ActionNotSupported);

}

return nullptr;

}

Var JavascriptOperators::NewJavascriptObjectNoArg(ScriptContext\* requestContext)

{

DynamicObject \* newObject = requestContext->GetLibrary()->CreateObject(true);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(newObject));

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

newObject = DynamicObject::FromVar(JavascriptProxy::AutoProxyWrapper(newObject));

}

#endif

return newObject;

}

Var JavascriptOperators::NewJavascriptArrayNoArg(ScriptContext\* requestContext)

{

JavascriptArray \* newArray = requestContext->GetLibrary()->CreateArray();

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(newArray));

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

newArray = static\_cast<JavascriptArray\*>(JavascriptProxy::AutoProxyWrapper(newArray));

}

#endif

return newArray;

}

Var JavascriptOperators::NewScObjectNoArgNoCtorFull(Var instance, ScriptContext\* requestContext)

{

return NewScObjectNoArgNoCtorCommon(instance, requestContext, true);

}

Var JavascriptOperators::NewScObjectNoArgNoCtor(Var instance, ScriptContext\* requestContext)

{

return NewScObjectNoArgNoCtorCommon(instance, requestContext, false);

}

Var JavascriptOperators::NewScObjectNoArgNoCtorCommon(Var instance, ScriptContext\* requestContext, bool isBaseClassConstructorNewScObject)

{

RecyclableObject \* object = RecyclableObject::FromVar(instance);

FunctionInfo\* functionInfo = JavascriptOperators::GetConstructorFunctionInfo(instance, requestContext);

Assert(functionInfo != &JavascriptObject::EntryInfo::NewInstance); // built-ins are not inlined

Assert(functionInfo != &JavascriptArray::EntryInfo::NewInstance); // built-ins are not inlined

return functionInfo != nullptr ?

JavascriptOperators::NewScObjectCommon(object, functionInfo, requestContext, isBaseClassConstructorNewScObject) :

JavascriptOperators::NewScObjectHostDispatchOrProxy(object, requestContext);

}

Var JavascriptOperators::NewScObjectNoArg(Var instance, ScriptContext \* requestContext)

{

if (JavascriptProxy::Is(instance))

{

Arguments args(CallInfo(CallFlags\_New, 1), &instance);

JavascriptProxy\* proxy = JavascriptProxy::FromVar(instance);

return proxy->ConstructorTrap(args, requestContext, 0);

}

FunctionInfo\* functionInfo = JavascriptOperators::GetConstructorFunctionInfo(instance, requestContext);

RecyclableObject \* object = RecyclableObject::FromVar(instance);

if (functionInfo == &JavascriptObject::EntryInfo::NewInstance)

{

// Fast path for new Object()

Assert((functionInfo->GetAttributes() & FunctionInfo::ErrorOnNew) == 0);

JavascriptLibrary\* library = object->GetLibrary();

DynamicObject \* newObject = library->CreateObject(true);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(newObject));

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

newObject = DynamicObject::FromVar(JavascriptProxy::AutoProxyWrapper(newObject));

}

#endif

#if DBG

DynamicType\* newObjectType = newObject->GetDynamicType();

Assert(newObjectType->GetIsShared());

JavascriptFunction\* constructor = JavascriptFunction::FromVar(instance);

Assert(!constructor->GetConstructorCache()->NeedsUpdateAfterCtor());

#endif

ScriptContext \* scriptContext = library->GetScriptContext();

if (scriptContext != requestContext)

{

CrossSite::MarshalDynamicObjectAndPrototype(requestContext, newObject);

}

return newObject;

}

else if (functionInfo == &JavascriptArray::EntryInfo::NewInstance)

{

Assert((functionInfo->GetAttributes() & FunctionInfo::ErrorOnNew) == 0);

JavascriptLibrary\* library = object->GetLibrary();

JavascriptArray \* newArray = library->CreateArray();

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(newArray));

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

newArray = static\_cast<JavascriptArray\*>(JavascriptProxy::AutoProxyWrapper(newArray));

}

#endif

#if DBG

DynamicType\* newArrayType = newArray->GetDynamicType();

Assert(newArrayType->GetIsShared());

JavascriptFunction\* constructor = JavascriptFunction::FromVar(instance);

Assert(!constructor->GetConstructorCache()->NeedsUpdateAfterCtor());

#endif

ScriptContext \* scriptContext = library->GetScriptContext();

if (scriptContext != requestContext)

{

CrossSite::MarshalDynamicObjectAndPrototype(requestContext, newArray);

}

return newArray;

}

Var newObject = functionInfo != nullptr ?

JavascriptOperators::NewScObjectCommon(object, functionInfo, requestContext) :

JavascriptOperators::NewScObjectHostDispatchOrProxy(object, requestContext);

Var returnVar = object->GetEntryPoint()(object, CallInfo(CallFlags\_New, 1), newObject);

if (JavascriptOperators::IsObject(returnVar))

{

newObject = returnVar;

}

ConstructorCache \* constructorCache = nullptr;

if (JavascriptFunction::Is(instance))

{

constructorCache = JavascriptFunction::FromVar(instance)->GetConstructorCache();

}

if (constructorCache != nullptr && constructorCache->NeedsUpdateAfterCtor())

{

JavascriptOperators::UpdateNewScObjectCache(object, newObject, requestContext);

}

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

newObject = DynamicObject::FromVar(JavascriptProxy::AutoProxyWrapper(newObject));

// this might come from a different scriptcontext.

newObject = CrossSite::MarshalVar(requestContext, newObject);

}

#endif

return newObject;

}

Var JavascriptOperators::NewScObjectNoCtorFull(Var instance, ScriptContext\* requestContext)

{

return NewScObjectNoCtorCommon(instance, requestContext, true);

}

Var JavascriptOperators::NewScObjectNoCtor(Var instance, ScriptContext \* requestContext)

{

return NewScObjectNoCtorCommon(instance, requestContext, false);

}

Var JavascriptOperators::NewScObjectNoCtorCommon(Var instance, ScriptContext\* requestContext, bool isBaseClassConstructorNewScObject)

{

FunctionInfo\* functionInfo = JavascriptOperators::GetConstructorFunctionInfo(instance, requestContext);

if (functionInfo)

{

return JavascriptOperators::NewScObjectCommon(RecyclableObject::FromVar(instance), functionInfo, requestContext, isBaseClassConstructorNewScObject);

}

else

{

return JavascriptOperators::NewScObjectHostDispatchOrProxy(RecyclableObject::FromVar(instance), requestContext);

}

}

Var JavascriptOperators::NewScObjectHostDispatchOrProxy(RecyclableObject \* function, ScriptContext \* requestContext)

{

ScriptContext\* functionScriptContext = function->GetScriptContext();

if (JavascriptProxy::Is(function))

{

// We can still call into NewScObjectNoCtor variations in JIT code for performance; however for proxy we don't

// really need the new object as the trap will handle the "this" pointer separately. pass back nullptr to ensure

// failure in invalid case.

return nullptr;

}

RecyclableObject \* prototype = JavascriptOperators::GetPrototypeObject(function, functionScriptContext);

prototype = RecyclableObject::FromVar(CrossSite::MarshalVar(requestContext, prototype));

Var object = requestContext->GetLibrary()->CreateObject(prototype);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(object));

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

object = DynamicObject::FromVar(JavascriptProxy::AutoProxyWrapper(object));

}

#endif

return object;

}

Var JavascriptOperators::NewScObjectCommon(RecyclableObject \* function, FunctionInfo\* functionInfo, ScriptContext \* requestContext, bool isBaseClassConstructorNewScObject)

{

// CONSIDER: Allow for the cache to be repopulated if the type got collected, and a new one got populated with

// the same number of inlined slots. This requires that the JIT-ed code actually load the type from the cache

// (instead of hard-coding it), but it can (and must) keep the hard-coded number of inline slots.

// CONSIDER: Consider also not pinning the type in the cache. This can be done by using a registration based

// weak reference (we need to control the memory address), which we don't yet have, or by allocating the cache from

// the inline cache arena to allow it to be zeroed, but retain a recycler-allocated portion to hold on to the size of

// inlined slots.

JavascriptFunction\* constructor = JavascriptFunction::FromVar(function);

if (functionInfo->IsClassConstructor() && !isBaseClassConstructorNewScObject)

{

// If we are calling new on a class constructor, the contract is that we pass new.target as the 'this' argument.

// function is the constructor on which we called new - which is new.target.

// If we are trying to construct the object for a base class constructor as part of a super call, we should not

// store new.target in the 'this' argument.

return function;

}

ConstructorCache\* constructorCache = constructor->GetConstructorCache();

AssertMsg(constructorCache->GetScriptContext() == nullptr || constructorCache->GetScriptContext() == constructor->GetScriptContext(),

"Why did we populate a constructor cache with a mismatched script context?");

Assert(constructorCache != nullptr);

DynamicType\* type = constructorCache->GetGuardValueAsType();

if (type != nullptr && constructorCache->GetScriptContext() == requestContext)

{

#if DBG

bool cachedProtoCanBeCached;

Assert(type->GetPrototype() == JavascriptOperators::GetPrototypeObjectForConstructorCache(constructor, requestContext, cachedProtoCanBeCached));

Assert(cachedProtoCanBeCached);

Assert(type->GetIsShared());

#endif

#if DBG\_DUMP

TraceUseConstructorCache(constructorCache, constructor, true);

#endif

Var object = DynamicObject::New(requestContext->GetRecycler(), type);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(object));

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

object = DynamicObject::FromVar(JavascriptProxy::AutoProxyWrapper(object));

}

#endif

return object;

}

if (constructorCache->SkipDefaultNewObject())

{

Assert(!constructorCache->NeedsUpdateAfterCtor());

#if DBG\_DUMP

TraceUseConstructorCache(constructorCache, constructor, true);

#endif

if (isBaseClassConstructorNewScObject)

{

return JavascriptOperators::CreateFromConstructor(function, requestContext);

}

return nullptr;

}

#if DBG\_DUMP

TraceUseConstructorCache(constructorCache, constructor, false);

#endif

ScriptContext\* constructorScriptContext = function->GetScriptContext();

Assert(!constructorScriptContext->GetThreadContext()->IsDisableImplicitException());

// we shouldn't try to call the constructor if it's closed already.

constructorScriptContext->VerifyAlive(TRUE, requestContext);

FunctionInfo::Attributes attributes = functionInfo->GetAttributes();

if (attributes & FunctionInfo::ErrorOnNew)

{

JavascriptError::ThrowTypeError(requestContext, JSERR\_ErrorOnNew);

}

// Slow path

FunctionProxy \* ctorProxy = constructor->GetFunctionProxy();

FunctionBody \* functionBody = ctorProxy != nullptr ? ctorProxy->EnsureDeserialized()->Parse() : nullptr;

if (attributes & FunctionInfo::SkipDefaultNewObject)

{

// The constructor doesn't use the default new object.

#pragma prefast(suppress:6236, "DevDiv bug 830883. False positive when PHASE\_OFF is #defined as '(false)'.")

if (!PHASE\_OFF1(ConstructorCachePhase) && (functionBody == nullptr || !PHASE\_OFF(ConstructorCachePhase, functionBody)))

{

constructorCache = constructor->EnsureValidConstructorCache();

constructorCache->PopulateForSkipDefaultNewObject(constructorScriptContext);

#if DBG\_DUMP

if ((functionBody != nullptr && PHASE\_TRACE(Js::ConstructorCachePhase, functionBody)) || (functionBody == nullptr && PHASE\_TRACE1(Js::ConstructorCachePhase)))

{

const wchar\_t\* ctorName = functionBody != nullptr ? functionBody->GetDisplayName() : L"<unknown>";

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"CtorCache: populated cache (0x%p) for ctor %s (%s): ", constructorCache, ctorName,

functionBody ? functionBody->GetDebugNumberSet(debugStringBuffer) : L"(null)");

constructorCache->Dump();

Output::Print(L"\n");

Output::Flush();

}

#endif

}

Assert(!constructorCache->NeedsUpdateAfterCtor());

return nullptr;

}

// CONSIDER: Create some form of PatchGetProtoObjForCtorCache, which actually caches the prototype object in the constructor cache.

// Make sure that it does NOT populate the guard field. On the slow path (the only path for cross-context calls) we can do a faster lookup

// after we fail the guard check. When invalidating the cache for proto change, make sure we zap the prototype field of the cache in

// addition to the guard value.

bool prototypeCanBeCached;

RecyclableObject\* prototype = JavascriptOperators::GetPrototypeObjectForConstructorCache(function, constructorScriptContext, prototypeCanBeCached);

prototype = RecyclableObject::FromVar(CrossSite::MarshalVar(requestContext, prototype));

DynamicObject\* newObject = requestContext->GetLibrary()->CreateObject(prototype, 8);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(newObject));

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

newObject = DynamicObject::FromVar(JavascriptProxy::AutoProxyWrapper(newObject));

}

#endif

Assert(newObject->GetTypeHandler()->GetPropertyCount() == 0);

if (prototypeCanBeCached && functionBody != nullptr && requestContext == constructorScriptContext &&

!Js::JavascriptProxy::Is(newObject) &&

!PHASE\_OFF1(ConstructorCachePhase) && !PHASE\_OFF(ConstructorCachePhase, functionBody))

{

DynamicType\* newObjectType = newObject->GetDynamicType();

// Initial type (without any properties) should always be shared up-front. This allows us to populate the cache right away.

Assert(newObjectType->GetIsShared());

// Populate the cache here and set the updateAfterCtor flag. This way, if the ctor is called recursively the

// recursive calls will hit the cache and use the initial type. On the unwind path, we will update the cache

// after the innermost ctor and clear the flag. After subsequent ctors we won't attempt an update anymore.

// As long as the updateAfterCtor flag is set it is safe to update the cache, because it would not have been

// hard-coded in the JIT-ed code.

constructorCache = constructor->EnsureValidConstructorCache();

constructorCache->Populate(newObjectType, constructorScriptContext, functionBody->GetHasNoExplicitReturnValue(), true);

Assert(constructorCache->IsConsistent());

#if DBG\_DUMP

if ((functionBody != nullptr && PHASE\_TRACE(Js::ConstructorCachePhase, functionBody)) || (functionBody == nullptr && PHASE\_TRACE1(Js::ConstructorCachePhase)))

{

const wchar\_t\* ctorName = functionBody != nullptr ? functionBody->GetDisplayName() : L"<unknown>";

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"CtorCache: populated cache (0x%p) for ctor %s (%s): ", constructorCache, ctorName,

functionBody ? functionBody->GetDebugNumberSet(debugStringBuffer) : L"(null)");

constructorCache->Dump();

Output::Print(L"\n");

Output::Flush();

}

#endif

}

else

{

#if DBG\_DUMP

if ((functionBody != nullptr && PHASE\_TRACE(Js::ConstructorCachePhase, functionBody)) || (functionBody == nullptr && PHASE\_TRACE1(Js::ConstructorCachePhase)))

{

const wchar\_t\* ctorName = functionBody != nullptr ? functionBody->GetDisplayName() : L"<unknown>";

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"CtorCache: did not populate cache (0x%p) for ctor %s (%s), because %s: prototype = 0x%p, functionBody = 0x%p, ctor context = 0x%p, request context = 0x%p",

constructorCache, ctorName, functionBody ? functionBody->GetDebugNumberSet(debugStringBuffer) : L"(null)",

!prototypeCanBeCached ? L"prototype cannot be cached" :

functionBody == nullptr ? L"function has no body" :

requestContext != constructorScriptContext ? L"of cross-context call" : L"constructor cache phase is off",

prototype, functionBody, constructorScriptContext, requestContext);

Output::Print(L"\n");

Output::Flush();

}

#endif

}

return newObject;

}

void JavascriptOperators::UpdateNewScObjectCache(Var function, Var instance, ScriptContext\* requestContext)

{

JavascriptFunction\* constructor = JavascriptFunction::FromVar(function);

if(constructor->GetScriptContext() != requestContext)

{

// The cache is populated only when the constructor function's context is the same as the calling context. However,

// the cached type is not finalized yet and may not be until multiple calls to the constructor have been made (see

// flag ConstructorCallsRequiredToFinalizeCachedType). A subsequent call to the constructor may be made from a

// different context, so ignore those cross-context calls and wait for the constructor to be called from its own

// context again to finalize the cached type.

return;

}

// Review : What happens if the cache got invalidated between NewScObject and here?

// Should we allocate new? Should we mark it as polymorphic?

ConstructorCache\* constructorCache = constructor->GetConstructorCache();

Assert(constructorCache->IsConsistent());

Assert(!ConstructorCache::IsDefault(constructorCache));

AssertMsg(constructorCache->GetScriptContext() == constructor->GetScriptContext(), "Why did we populate a constructor cache with a mismatched script context?");

AssertMsg(constructorCache->IsPopulated(), "Why are we updating a constructor cache that hasn't been populated?");

// The presence of the updateAfterCtor flag guarantees that this cache hasn't been used in JIT-ed fast path. Even, if the

// cache is invalidated, this flag is not changed.

AssertMsg(constructorCache->NeedsUpdateAfterCtor(), "Why are we updating a constructor cache that doesn't need to be updated?");

const bool finalizeCachedType =

constructorCache->CallCount() >= CONFIG\_FLAG(ConstructorCallsRequiredToFinalizeCachedType);

if(!finalizeCachedType)

{

constructorCache->IncCallCount();

}

else

{

constructorCache->ClearUpdateAfterCtor();

}

FunctionBody\* constructorBody = constructor->GetFunctionBody();

AssertMsg(constructorBody != nullptr, "Constructor function doesn't have a function body.");

Assert(RecyclableObject::Is(instance));

// The cache might have been invalidated between NewScObjectCommon and UpdateNewScObjectCache. This could occur, for example, if

// the constructor updates its own prototype property. If that happens we don't want to re-populate it here. A new cache will

// be created when the constructor is called again.

if (constructorCache->IsInvalidated())

{

#if DBG\_DUMP

TraceUpdateConstructorCache(constructorCache, constructorBody, false, L"because cache is invalidated");

#endif

return;

}

Assert(constructorCache->GetGuardValueAsType() != nullptr);

if (DynamicType::Is(RecyclableObject::FromVar(instance)->GetTypeId()))

{

DynamicObject \*object = DynamicObject::FromVar(instance);

DynamicType\* type = object->GetDynamicType();

DynamicTypeHandler\* typeHandler = type->GetTypeHandler();

if (constructorBody->GetHasOnlyThisStmts())

{

if (typeHandler->IsSharable())

{

#if DBG

bool cachedProtoCanBeCached;

Assert(type->GetPrototype() == JavascriptOperators::GetPrototypeObjectForConstructorCache(constructor, requestContext, cachedProtoCanBeCached));

Assert(cachedProtoCanBeCached);

Assert(type->GetScriptContext() == constructorCache->GetScriptContext());

Assert(type->GetPrototype() == constructorCache->GetType()->GetPrototype());

#endif

typeHandler->SetMayBecomeShared();

// CONSIDER: Remove only this for delayed type sharing.

type->ShareType();

#if ENABLE\_PROFILE\_INFO

DynamicProfileInfo\* profileInfo = constructorBody->HasDynamicProfileInfo() ? constructorBody->GetAnyDynamicProfileInfo() : nullptr;

if ((profileInfo != nullptr && profileInfo->GetImplicitCallFlags() <= ImplicitCall\_None) ||

CheckIfPrototypeChainHasOnlyWritableDataProperties(type->GetPrototype()))

{

Assert(typeHandler->GetPropertyCount() < Js::PropertyIndexRanges<PropertyIndex>::MaxValue);

for (PropertyIndex pi = 0; pi < typeHandler->GetPropertyCount(); pi++)

{

requestContext->RegisterConstructorCache(typeHandler->GetPropertyId(requestContext, pi), constructorCache);

}

Assert(constructorBody->GetUtf8SourceInfo()->GetIsLibraryCode() || !constructor->GetScriptContext()->IsInDebugMode());

if (constructorCache->TryUpdateAfterConstructor(type, constructor->GetScriptContext()))

{

#if DBG\_DUMP

TraceUpdateConstructorCache(constructorCache, constructorBody, true, L"");

#endif

}

else

{

#if DBG\_DUMP

TraceUpdateConstructorCache(constructorCache, constructorBody, false, L"because number of slots > MaxCachedSlotCount");

#endif

}

}

#if DBG\_DUMP

else

{

if (profileInfo &&

((profileInfo->GetImplicitCallFlags() & ~(Js::ImplicitCall\_External | Js::ImplicitCall\_Accessor)) == 0) &&

profileInfo != nullptr && CheckIfPrototypeChainHasOnlyWritableDataProperties(type->GetPrototype()) &&

Js::Configuration::Global.flags.Trace.IsEnabled(Js::HostOptPhase))

{

const wchar\_t\* ctorName = constructorBody->GetDisplayName();

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"CtorCache: %s cache (0x%p) for ctor %s (#%u) did not update because external call",

constructorCache, constructorBody, ctorName, constructorBody ? constructorBody->GetDebugNumberSet(debugStringBuffer) : L"(null)");

Output::Print(L"\n");

Output::Flush();

}

}

#endif

#endif

}

else

{

// Dynamic type created is not sharable.

// So in future don't try to check for "this assignment optimization".

constructorBody->SetHasOnlyThisStmts(false);

#if DBG\_DUMP

TraceUpdateConstructorCache(constructorCache, constructorBody, false, L"because final type is not shareable");

#endif

}

}

else

{

#if DBG\_DUMP

TraceUpdateConstructorCache(constructorCache, constructorBody, false, L"because ctor has not only this statements");

#endif

}

}

else

{

// Even though this constructor apparently returned something other than the default object we created,

// it still makes sense to cache the parameters of the default object, since we must create it every time, anyway.

#if DBG\_DUMP

TraceUpdateConstructorCache(constructorCache, constructorBody, false, L"because ctor return a non-object value");

#endif

return;

}

// Whatever the constructor returned, if we're caching a type we want to be sure we shrink its inline slot capacity.

if (finalizeCachedType && constructorCache->IsEnabled())

{

DynamicType\* cachedType = constructorCache->NeedsTypeUpdate() ? constructorCache->GetPendingType() : constructorCache->GetType();

DynamicTypeHandler\* cachedTypeHandler = cachedType->GetTypeHandler();

// Consider: We could delay inline slot capacity shrinking until the second time this constructor is invoked. In some cases

// this might permit more properties to remain inlined if the objects grow after constructor. This would require flagging

// the cache as special (already possible) and forcing the shrinking during work item creation if we happen to JIT this

// constructor while the cache is in this special state.

if (cachedTypeHandler->GetInlineSlotCapacity())

{

#if DBG\_DUMP

int inlineSlotCapacityBeforeShrink = cachedTypeHandler->GetInlineSlotCapacity();

#endif

// Note that after the cache has been updated and might have been used in the JIT-ed code, it is no longer legal to

// shrink the inline slot capacity of the type. That's because we allocate memory for a fixed number of inlined properties

// and if that number changed on the type, this update wouldn't get reflected in JIT-ed code and we would allocate objects

// of a wrong size. This could conceivably happen if the original object got collected, and with it some of the successor

// types also. If then another constructor has the same prototype and needs to populate its own cache, it would attempt to

// shrink inlined slots again. If all surviving type handlers have smaller inline slot capacity, we would shrink it further.

// To address this problem the type handler has a bit indicating its inline slots have been shrunk already. If that bit is

// set ShrinkSlotAndInlineSlotCapacity does nothing.

cachedTypeHandler->ShrinkSlotAndInlineSlotCapacity();

constructorCache->UpdateInlineSlotCount();

#if DBG\_DUMP

Assert(inlineSlotCapacityBeforeShrink >= cachedTypeHandler->GetInlineSlotCapacity());

if (Js::Configuration::Global.flags.Trace.IsEnabled(Js::InlineSlotsPhase))

{

if (inlineSlotCapacityBeforeShrink != cachedTypeHandler->GetInlineSlotCapacity())

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"Inline slot capacity shrunk: Function:%04s Before:%d After:%d\n",

constructorBody->GetDebugNumberSet(debugStringBuffer), inlineSlotCapacityBeforeShrink, cachedTypeHandler->GetInlineSlotCapacity());

}

}

#endif

}

}

}

void JavascriptOperators::TraceUseConstructorCache(const ConstructorCache\* ctorCache, const JavascriptFunction\* ctor, bool isHit)

{

#if DBG\_DUMP

// We are under debug, so we can incur the extra check here.

FunctionProxy\* ctorBody = ctor->GetFunctionProxy();

if (ctorBody != nullptr && !ctorBody->GetScriptContext()->IsClosed())

{

ctorBody = ctorBody->EnsureDeserialized();

}

if ((ctorBody != nullptr && PHASE\_TRACE(Js::ConstructorCachePhase, ctorBody)) || (ctorBody == nullptr && PHASE\_TRACE1(Js::ConstructorCachePhase)))

{

const wchar\_t\* ctorName = ctorBody != nullptr ? ctorBody->GetDisplayName() : L"<unknown>";

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"CtorCache: %s cache (0x%p) for ctor %s (%s): ", isHit ? L"hit" : L"missed", ctorCache, ctorName,

ctorBody ? ctorBody->GetDebugNumberSet(debugStringBuffer) : L"(null)");

ctorCache->Dump();

Output::Print(L"\n");

Output::Flush();

}

#endif

}

void JavascriptOperators::TraceUpdateConstructorCache(const ConstructorCache\* ctorCache, const FunctionBody\* ctorBody, bool updated, const wchar\_t\* reason)

{

#if DBG\_DUMP

if (PHASE\_TRACE(Js::ConstructorCachePhase, ctorBody))

{

const wchar\_t\* ctorName = ctorBody->GetDisplayName();

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"CtorCache: %s cache (0x%p) for ctor %s (%s)%s %s: ",

updated ? L"updated" : L"did not update", ctorBody, ctorName,

ctorBody ? const\_cast<Js::FunctionBody \*>(ctorBody)->GetDebugNumberSet(debugStringBuffer) : L"(null)",

updated ? L"" : L", because" , reason);

ctorCache->Dump();

Output::Print(L"\n");

Output::Flush();

}

#endif

}

Var JavascriptOperators::NewScObject(const Var callee, const Arguments args, ScriptContext \*const scriptContext, const Js::AuxArray<uint32> \*spreadIndices)

{

Assert(callee);

Assert(args.Info.Count != 0);

Assert(scriptContext);

// Always save and restore implicit call flags when calling out

// REVIEW: Can we avoid it if we don't collect dynamic profile info?

ThreadContext \*const threadContext = scriptContext->GetThreadContext();

const ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

const Var newVarInstance = JavascriptFunction::CallAsConstructor(callee, /\* overridingNewTarget = \*/nullptr, args, scriptContext, spreadIndices);

threadContext->SetImplicitCallFlags(savedImplicitCallFlags);

return newVarInstance;

}

Js::GlobalObject \* JavascriptOperators::OP\_LdRoot(ScriptContext\* scriptContext)

{

return scriptContext->GetGlobalObject();

}

Js::ModuleRoot \* JavascriptOperators::GetModuleRoot(int moduleID, ScriptContext\* scriptContext)

{

Assert(moduleID != kmodGlobal);

JavascriptLibrary\* library = scriptContext->GetLibrary();

HostObjectBase \*hostObject = library->GetGlobalObject()->GetHostObject();

if (hostObject)

{

Js::ModuleRoot \* moduleRoot = hostObject->GetModuleRoot(moduleID);

Assert(!CrossSite::NeedMarshalVar(moduleRoot, scriptContext));

return moduleRoot;

}

HostScriptContext \*hostScriptContext = scriptContext->GetHostScriptContext();

if (hostScriptContext)

{

Js::ModuleRoot \* moduleRoot = hostScriptContext->GetModuleRoot(moduleID);

Assert(!CrossSite::NeedMarshalVar(moduleRoot, scriptContext));

return moduleRoot;

}

Assert(FALSE);

return nullptr;

}

Var JavascriptOperators::OP\_LoadModuleRoot(int moduleID, ScriptContext\* scriptContext)

{

Js::ModuleRoot \* moduleRoot = GetModuleRoot(moduleID, scriptContext);

if (moduleRoot)

{

return moduleRoot;

}

Assert(false);

return scriptContext->GetLibrary()->GetUndefined();

}

Var JavascriptOperators::OP\_LdNull(ScriptContext\* scriptContext)

{

return scriptContext->GetLibrary()->GetNull();

}

Var JavascriptOperators::OP\_LdUndef(ScriptContext\* scriptContext)

{

return scriptContext->GetLibrary()->GetUndefined();

}

Var JavascriptOperators::OP\_LdNaN(ScriptContext\* scriptContext)

{

return scriptContext->GetLibrary()->GetNaN();

}

Var JavascriptOperators::OP\_LdInfinity(ScriptContext\* scriptContext)

{

return scriptContext->GetLibrary()->GetPositiveInfinite();

}

void JavascriptOperators::BuildHandlerScope(Var argThis, RecyclableObject \* hostObject, FrameDisplay \* pDisplay, ScriptContext \* scriptContext)

{

Assert(argThis != nullptr);

pDisplay->SetItem(0, TaggedNumber::Is(argThis) ? scriptContext->GetLibrary()->CreateNumberObject(argThis) : argThis);

uint16 i = 1;

Var aChild = argThis;

uint16 length = pDisplay->GetLength();

// Now add any parent scopes

// We need to support the namespace parent lookup in both fastDOM on and off scenario.

while (aChild != NULL)

{

Var aParent = hostObject->GetNamespaceParent(aChild);

if (aParent == nullptr)

{

break;

}

aParent = CrossSite::MarshalVar(scriptContext, aParent);

if (i == length)

{

length += 8;

FrameDisplay \* tmp = RecyclerNewPlus(scriptContext->GetRecycler(), length \* sizeof(void\*), FrameDisplay, length);

js\_memcpy\_s((char\*)tmp + tmp->GetOffsetOfScopes(), tmp->GetLength() \* sizeof(void \*), (char\*)pDisplay + pDisplay->GetOffsetOfScopes(), pDisplay->GetLength() \* sizeof(void\*));

pDisplay = tmp;

}

pDisplay->SetItem(i, aParent);

aChild = aParent;

i++;

}

Assert(i <= pDisplay->GetLength());

pDisplay->SetLength(i);

}

FrameDisplay \* JavascriptOperators::OP\_LdHandlerScope(Var argThis, ScriptContext\* scriptContext)

{

// The idea here is to build a stack of nested scopes in the form of a JS array.

//

// The scope stack for an event handler looks like this:

//

// implicit "this"

// implicit namespace parent scopes

// Put the implicit "this"

if (argThis != NULL)

{

RecyclableObject\* hostObject = scriptContext->GetGlobalObject()->GetHostObject();

if (hostObject == nullptr)

{

hostObject = scriptContext->GetGlobalObject()->GetDirectHostObject();

}

if (hostObject != nullptr)

{

uint16 length = 7;

FrameDisplay \*pDisplay =

RecyclerNewPlus(scriptContext->GetRecycler(), length \* sizeof(void\*), FrameDisplay, length);

BuildHandlerScope(argThis, hostObject, pDisplay, scriptContext);

return pDisplay;

}

}

return const\_cast<FrameDisplay \*>(&Js::NullFrameDisplay);

}

FrameDisplay\* JavascriptOperators::OP\_LdFrameDisplay(void \*argHead, void \*argEnv, ScriptContext\* scriptContext)

{

// Build a display of nested frame objects.

// argHead is the current scope; argEnv is either the lone trailing scope or an array of scopes

// which we append to the new display.

// Note that there are cases in which a function with no local frame must construct a display to pass

// to the function(s) nested within it. In such a case, argHead will be a null object, and it's not

// strictly necessary to include it. But such cases are rare and not perf critical, so it's not

// worth the extra complexity to notify the nested functions that they can "skip" this slot in the

// frame display when they're loading scopes nested outside it.

FrameDisplay \*pDisplay = nullptr;

FrameDisplay \*envDisplay = (FrameDisplay\*)argEnv;

uint16 length = envDisplay->GetLength() + 1;

pDisplay = RecyclerNewPlus(scriptContext->GetRecycler(), length \* sizeof(void\*), FrameDisplay, length);

for (int j = 0; j < length - 1; j++)

{

pDisplay->SetItem(j + 1, envDisplay->GetItem(j));

}

pDisplay->SetItem(0, argHead);

return pDisplay;

}

FrameDisplay\* JavascriptOperators::OP\_LdFrameDisplayNoParent(void \*argHead, ScriptContext\* scriptContext)

{

return OP\_LdFrameDisplay(argHead, (void\*)&NullFrameDisplay, scriptContext);

}

FrameDisplay\* JavascriptOperators::OP\_LdStrictFrameDisplay(void \*argHead, void \*argEnv, ScriptContext\* scriptContext)

{

FrameDisplay \* pDisplay = OP\_LdFrameDisplay(argHead, argEnv, scriptContext);

pDisplay->SetStrictMode(true);

return pDisplay;

}

FrameDisplay\* JavascriptOperators::OP\_LdStrictFrameDisplayNoParent(void \*argHead, ScriptContext\* scriptContext)

{

return OP\_LdStrictFrameDisplay(argHead, (void\*)&StrictNullFrameDisplay, scriptContext);

}

FrameDisplay\* JavascriptOperators::OP\_LdInnerFrameDisplay(void \*argHead, void \*argEnv, ScriptContext\* scriptContext)

{

CheckInnerFrameDisplayArgument(argHead);

return OP\_LdFrameDisplay(argHead, argEnv, scriptContext);

}

FrameDisplay\* JavascriptOperators::OP\_LdInnerFrameDisplayNoParent(void \*argHead, ScriptContext\* scriptContext)

{

CheckInnerFrameDisplayArgument(argHead);

return OP\_LdFrameDisplayNoParent(argHead, scriptContext);

}

FrameDisplay\* JavascriptOperators::OP\_LdStrictInnerFrameDisplay(void \*argHead, void \*argEnv, ScriptContext\* scriptContext)

{

CheckInnerFrameDisplayArgument(argHead);

return OP\_LdStrictFrameDisplay(argHead, argEnv, scriptContext);

}

FrameDisplay\* JavascriptOperators::OP\_LdStrictInnerFrameDisplayNoParent(void \*argHead, ScriptContext\* scriptContext)

{

CheckInnerFrameDisplayArgument(argHead);

return OP\_LdStrictFrameDisplayNoParent(argHead, scriptContext);

}

void JavascriptOperators::CheckInnerFrameDisplayArgument(void \*argHead)

{

if (ThreadContext::IsOnStack(argHead))

{

AssertMsg(false, "Illegal byte code: stack object as with scope");

Js::Throw::FatalInternalError();

}

if (!RecyclableObject::Is(argHead))

{

AssertMsg(false, "Illegal byte code: non-object as with scope");

Js::Throw::FatalInternalError();

}

}

Js::PropertyId JavascriptOperators::GetPropertyId(Var propertyName, ScriptContext\* scriptContext)

{

PropertyRecord const \* propertyRecord = nullptr;

if (JavascriptSymbol::Is(propertyName))

{

propertyRecord = JavascriptSymbol::FromVar(propertyName)->GetValue();

}

else if (JavascriptSymbolObject::Is(propertyName))

{

propertyRecord = JavascriptSymbolObject::FromVar(propertyName)->GetValue();

}

else

{

JavascriptString \* indexStr = JavascriptConversion::ToString(propertyName, scriptContext);

scriptContext->GetOrAddPropertyRecord(indexStr->GetString(), indexStr->GetLength(), &propertyRecord);

}

return propertyRecord->GetPropertyId();

}

void JavascriptOperators::OP\_InitSetter(Var object, PropertyId propertyId, Var setter)

{

AssertMsg(!TaggedNumber::Is(object), "SetMember on a non-object?");

RecyclableObject::FromVar(object)->SetAccessors(propertyId, nullptr, setter);

}

void JavascriptOperators::OP\_InitClassMemberSet(Var object, PropertyId propertyId, Var setter)

{

JavascriptOperators::OP\_InitSetter(object, propertyId, setter);

RecyclableObject::FromVar(object)->SetAttributes(propertyId, PropertyClassMemberDefaults);

}

Js::PropertyId JavascriptOperators::OP\_InitElemSetter(Var object, Var elementName, Var setter, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

AssertMsg(!TaggedNumber::Is(object), "SetMember on a non-object?");

PropertyId propertyId = JavascriptOperators::GetPropertyId(elementName, scriptContext);

RecyclableObject::FromVar(object)->SetAccessors(propertyId, nullptr, setter);

return propertyId;

}

void JavascriptOperators::OP\_InitClassMemberSetComputedName(Var object, Var elementName, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

Js::PropertyId propertyId = JavascriptOperators::OP\_InitElemSetter(object, elementName, value, scriptContext);

RecyclableObject\* instance = RecyclableObject::FromVar(object);

// instance will be a function if it is the class constructor (otherwise it would be an object)

if (JavascriptFunction::Is(instance) && Js::PropertyIds::prototype == propertyId)

{

// It is a TypeError to have a static member with a computed name that evaluates to 'prototype'

JavascriptError::ThrowTypeError(scriptContext, JSERR\_ClassStaticMethodCannotBePrototype);

}

instance->SetAttributes(propertyId, PropertyClassMemberDefaults);

}

BOOL JavascriptOperators::IsClassConstructor(Var instance)

{

return JavascriptFunction::Is(instance) && (JavascriptFunction::FromVar(instance)->GetFunctionInfo()->IsClassConstructor() || !JavascriptFunction::FromVar(instance)->IsScriptFunction());

}

void JavascriptOperators::OP\_InitGetter(Var object, PropertyId propertyId, Var getter)

{

AssertMsg(!TaggedNumber::Is(object), "GetMember on a non-object?");

RecyclableObject::FromVar(object)->SetAccessors(propertyId, getter, nullptr);

}

void JavascriptOperators::OP\_InitClassMemberGet(Var object, PropertyId propertyId, Var getter)

{

JavascriptOperators::OP\_InitGetter(object, propertyId, getter);

RecyclableObject::FromVar(object)->SetAttributes(propertyId, PropertyClassMemberDefaults);

}

Js::PropertyId JavascriptOperators::OP\_InitElemGetter(Var object, Var elementName, Var getter, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

AssertMsg(!TaggedNumber::Is(object), "GetMember on a non-object?");

PropertyId propertyId = JavascriptOperators::GetPropertyId(elementName, scriptContext);

RecyclableObject::FromVar(object)->SetAccessors(propertyId, getter, nullptr);

return propertyId;

}

void JavascriptOperators::OP\_InitClassMemberGetComputedName(Var object, Var elementName, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

Js::PropertyId propertyId = JavascriptOperators::OP\_InitElemGetter(object, elementName, value, scriptContext);

RecyclableObject\* instance = RecyclableObject::FromVar(object);

// instance will be a function if it is the class constructor (otherwise it would be an object)

if (JavascriptFunction::Is(instance) && Js::PropertyIds::prototype == propertyId)

{

// It is a TypeError to have a static member with a computed name that evaluates to 'prototype'

JavascriptError::ThrowTypeError(scriptContext, JSERR\_ClassStaticMethodCannotBePrototype);

}

instance->SetAttributes(propertyId, PropertyClassMemberDefaults);

}

void JavascriptOperators::OP\_InitComputedProperty(Var object, Var elementName, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

PropertyId propertyId = JavascriptOperators::GetPropertyId(elementName, scriptContext);

RecyclableObject::FromVar(object)->InitProperty(propertyId, value, flags);

}

void JavascriptOperators::OP\_InitClassMemberComputedName(Var object, Var elementName, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

PropertyId propertyId = JavascriptOperators::GetPropertyId(elementName, scriptContext);

RecyclableObject\* instance = RecyclableObject::FromVar(object);

// instance will be a function if it is the class constructor (otherwise it would be an object)

if (JavascriptFunction::Is(instance) && Js::PropertyIds::prototype == propertyId)

{

// It is a TypeError to have a static member with a computed name that evaluates to 'prototype'

JavascriptError::ThrowTypeError(scriptContext, JSERR\_ClassStaticMethodCannotBePrototype);

}

instance->SetPropertyWithAttributes(propertyId, value, PropertyClassMemberDefaults, NULL, flags);

}

//

// Used by object literal {..., \_\_proto\_\_: ..., }.

// When \_\_proto\_\_ is enabled, it is effectively same as StFld. However when \_\_proto\_\_ is disabled, it functions same as InitFld.

//

void JavascriptOperators::OP\_InitProto(Var instance, PropertyId propertyId, Var value)

{

AssertMsg(RecyclableObject::Is(instance), "\_\_proto\_\_ member on a non-object?");

Assert(propertyId == PropertyIds::\_\_proto\_\_);

RecyclableObject\* object = RecyclableObject::FromVar(instance);

ScriptContext\* scriptContext = object->GetScriptContext();

if (scriptContext->GetConfig()->Is\_\_proto\_\_Enabled())

{

// B.3.1 \_\_proto\_\_\_ Property Names in Object Initializers

//6.If propKey is the string value "\_\_proto\_\_" and if isComputedPropertyName(propKey) is false, then

// a.If Type(v) is either Object or Null, then

// i.Return the result of calling the [[SetInheritance]] internal method of object with argument propValue.

// b.Return NormalCompletion(empty).

if (JavascriptOperators::IsObjectOrNull(value))

{

JavascriptObject::ChangePrototype(object, RecyclableObject::FromVar(value), /\*validate\*/false, scriptContext);

}

}

else

{

object->InitProperty(propertyId, value);

}

}

Var JavascriptOperators::ConvertToUnmappedArguments(HeapArgumentsObject \*argumentsObject,

uint32 paramCount,

Var \*paramAddr,

DynamicObject\* frameObject,

Js::PropertyIdArray \*propIds,

uint32 formalsCount,

ScriptContext\* scriptContext)

{

Var \*paramIter = paramAddr;

uint32 i = 0;

for (paramIter = paramAddr + i; i < paramCount; i++, paramIter++)

{

JavascriptOperators::SetItem(argumentsObject, argumentsObject, i, \*paramIter, scriptContext, PropertyOperation\_None, /\* skipPrototypeCheck = \*/ TRUE);

}

argumentsObject = argumentsObject->ConvertToUnmappedArgumentsObject();

// Now as the unmapping is done we need to fill those frame object with Undecl

for (i = 0; i < formalsCount; i++)

{

frameObject->SetSlot(SetSlotArguments(propIds != nullptr ? propIds->elements[i] : Js::Constants::NoProperty, i, scriptContext->GetLibrary()->GetUndeclBlockVar()));

}

return argumentsObject;

}

Var JavascriptOperators::LoadHeapArguments(JavascriptFunction \*funcCallee, uint32 paramCount, Var \*paramAddr, Var frameObj, Var vArray, ScriptContext\* scriptContext, bool nonSimpleParamList)

{

AssertMsg(paramCount != (unsigned int)-1, "Loading the arguments object in the global function?");

// Create and initialize the Arguments object.

uint32 formalsCount = 0;

Js::PropertyIdArray \*propIds = nullptr;

if (vArray != scriptContext->GetLibrary()->GetNull())

{

propIds = (Js::PropertyIdArray \*)vArray;

formalsCount = propIds->count;

}

HeapArgumentsObject \*argsObj = JavascriptOperators::CreateHeapArguments(funcCallee, paramCount, formalsCount, frameObj, scriptContext);

// Transfer formal arguments (that were actually passed) from their ArgIn slots to the local frame object.

uint32 i;

Var \*tmpAddr = paramAddr;

if (propIds != nullptr)

{

ActivationObject\* frameObject = (ActivationObject\*)frameObj;

// No fixed fields for formal parameters of the arguments object. Also, mark all fields as initialized up-front, because

// we will set them directly using SetSlot below, so the type handler will not have a chance to mark them as initialized later.

// CONSIDER : When we delay type sharing until the second instance is created, pass an argument indicating we want the types

// and handlers created here to be marked as shared up-front. This is to ensure we don't get any fixed fields and that the handler

// is ready for storing values directly to slots.

DynamicType\* newType = PathTypeHandlerBase::CreateNewScopeObject(scriptContext, frameObject->GetDynamicType(), propIds, nonSimpleParamList ? PropertyLetDefaults : PropertyNone);

int oldSlotCapacity = frameObject->GetDynamicType()->GetTypeHandler()->GetSlotCapacity();

int newSlotCapacity = newType->GetTypeHandler()->GetSlotCapacity();

\_\_analysis\_assume((uint32)newSlotCapacity >= formalsCount);

frameObject->EnsureSlots(oldSlotCapacity, newSlotCapacity, scriptContext, newType->GetTypeHandler());

frameObject->ReplaceType(newType);

if (nonSimpleParamList)

{

return ConvertToUnmappedArguments(argsObj, paramCount, paramAddr, frameObject, propIds, formalsCount, scriptContext);

}

for (i = 0; i < formalsCount && i < paramCount; i++, tmpAddr++)

{

frameObject->SetSlot(SetSlotArguments(propIds->elements[i], i, \*tmpAddr));

}

if (i < formalsCount)

{

// The formals that weren't passed still need to be put in the frame object so that

// their names will be found. Initialize them to "undefined".

for (; i < formalsCount; i++)

{

frameObject->SetSlot(SetSlotArguments(propIds->elements[i], i, scriptContext->GetLibrary()->GetUndefined()));

}

}

}

// Transfer the unnamed actual arguments, if any, to the Arguments object itself.

for (i = formalsCount, tmpAddr = paramAddr + i; i < paramCount; i++, tmpAddr++)

{

// ES5 10.6.11: use [[DefineOwnProperty]] semantics (instead of [[Put]]):

// do not check whether property is non-writable/etc in the prototype.

// ES3 semantics is same.

JavascriptOperators::SetItem(argsObj, argsObj, i, \*tmpAddr, scriptContext, PropertyOperation\_None, /\* skipPrototypeCheck = \*/ TRUE);

}

if (funcCallee->IsStrictMode())

{

// If the formals are let decls, then we just overwrote the frame object slots with

// Undecl sentinels, and we can use the original arguments that were passed to the HeapArgumentsObject.

return argsObj->ConvertToUnmappedArgumentsObject(!nonSimpleParamList);

}

return argsObj;

}

Var JavascriptOperators::LoadHeapArgsCached(JavascriptFunction \*funcCallee, uint32 actualsCount, uint32 formalsCount, Var \*paramAddr, Var frameObj, ScriptContext\* scriptContext, bool nonSimpleParamList)

{

// Disregard the "this" param.

AssertMsg(actualsCount != (uint32)-1 && formalsCount != (uint32)-1,

"Loading the arguments object in the global function?");

// Create and initialize the Arguments object.

HeapArgumentsObject \*argsObj = JavascriptOperators::CreateHeapArguments(funcCallee, actualsCount, formalsCount, frameObj, scriptContext);

// Transfer formal arguments (that were actually passed) from their ArgIn slots to the local frame object.

uint32 i;

Var \*tmpAddr = paramAddr;

if (formalsCount != 0)

{

DynamicObject\* frameObject = DynamicObject::FromVar(frameObj);

\_\_analysis\_assume((uint32)frameObject->GetDynamicType()->GetTypeHandler()->GetSlotCapacity() >= formalsCount);

if (nonSimpleParamList)

{

return ConvertToUnmappedArguments(argsObj, actualsCount, paramAddr, frameObject, nullptr /\*propIds\*/, formalsCount, scriptContext);

}

for (i = 0; i < formalsCount && i < actualsCount; i++, tmpAddr++)

{

// We don't know the propertyId at this point.

frameObject->SetSlot(SetSlotArguments(Constants::NoProperty, i, \*tmpAddr));

}

if (i < formalsCount)

{

// The formals that weren't passed still need to be put in the frame object so that

// their names will be found. Initialize them to "undefined".

for (; i < formalsCount; i++)

{

// We don't know the propertyId at this point.

frameObject->SetSlot(SetSlotArguments(Constants::NoProperty, i, scriptContext->GetLibrary()->GetUndefined()));

}

}

}

// Transfer the unnamed actual arguments, if any, to the Arguments object itself.

for (i = formalsCount, tmpAddr = paramAddr + i; i < actualsCount; i++, tmpAddr++)

{

// ES5 10.6.11: use [[DefineOwnProperty]] semantics (instead of [[Put]]):

// do not check whether property is non-writable/etc in the prototype.

// ES3 semantics is same.

JavascriptOperators::SetItem(argsObj, argsObj, i, \*tmpAddr, scriptContext, PropertyOperation\_None, /\* skipPrototypeCheck = \*/ TRUE);

}

if (funcCallee->IsStrictMode())

{

// If the formals are let decls, then we just overwrote the frame object slots with

// Undecl sentinels, and we can use the original arguments that were passed to the HeapArgumentsObject.

return argsObj->ConvertToUnmappedArgumentsObject(!nonSimpleParamList);

}

return argsObj;

}

HeapArgumentsObject \*JavascriptOperators::CreateHeapArguments(JavascriptFunction \*funcCallee, uint32 actualsCount, uint32 formalsCount, Var frameObj, ScriptContext\* scriptContext)

{

JavascriptLibrary \*library = scriptContext->GetLibrary();

HeapArgumentsObject \*argsObj = library->CreateHeapArguments(frameObj, formalsCount);

//

// Set the number of arguments of Arguments Object

//

argsObj->SetNumberOfArguments(actualsCount);

JavascriptOperators::SetProperty(argsObj, argsObj, PropertyIds::length, JavascriptNumber::ToVar(actualsCount, scriptContext), scriptContext);

if (scriptContext->GetConfig()->IsES6IteratorsEnabled())

{

JavascriptOperators::SetProperty(argsObj, argsObj, PropertyIds::\_symbolIterator, library->GetArrayPrototypeValuesFunction(), scriptContext);

}

if (funcCallee->IsStrictMode())

{

PropertyDescriptor propertyDescriptorCaller;

JavascriptFunction\* callerAccessor = library->GetThrowTypeErrorCallerAccessorFunction();

propertyDescriptorCaller.SetGetter(callerAccessor);

propertyDescriptorCaller.SetSetter(callerAccessor);

propertyDescriptorCaller.SetEnumerable(false);

propertyDescriptorCaller.SetConfigurable(false);

argsObj->SetAccessors(PropertyIds::caller, callerAccessor, callerAccessor, PropertyOperation\_NonFixedValue);

JavascriptOperators::SetAttributes(argsObj, PropertyIds::caller, propertyDescriptorCaller, false);

PropertyDescriptor propertyDescriptorCallee;

JavascriptFunction\* calleeAccessor = library->GetThrowTypeErrorCalleeAccessorFunction();

propertyDescriptorCallee.SetGetter(calleeAccessor);

propertyDescriptorCallee.SetSetter(calleeAccessor);

propertyDescriptorCallee.SetEnumerable(false);

propertyDescriptorCallee.SetConfigurable(false);

argsObj->SetAccessors(PropertyIds::callee, calleeAccessor, calleeAccessor, PropertyOperation\_NonFixedValue);

JavascriptOperators::SetAttributes(argsObj, PropertyIds::callee, propertyDescriptorCallee, false);

}

else

{

JavascriptOperators::SetProperty(argsObj, argsObj, PropertyIds::callee,

StackScriptFunction::EnsureBoxed(BOX\_PARAM(funcCallee, nullptr, L"callee")), scriptContext);

}

return argsObj;

}

Var JavascriptOperators::OP\_NewScopeObject(ScriptContext\*scriptContext)

{

return scriptContext->GetLibrary()->CreateActivationObject();

}

Var\* JavascriptOperators::OP\_NewScopeSlots(unsigned int size, ScriptContext \*scriptContext, Var scope)

{

Assert(size > ScopeSlots::FirstSlotIndex); // Should never see empty slot array

Var\* slotArray = RecyclerNewArray(scriptContext->GetRecycler(), Var, size); // last initialized slot contains reference to array of propertyIds, correspondent to objects in previous slots

uint count = size - ScopeSlots::FirstSlotIndex;

ScopeSlots slots(slotArray);

slots.SetCount(count);

slots.SetScopeMetadata(scope);

Var undef = scriptContext->GetLibrary()->GetUndefined();

for (unsigned int i = 0; i < count; i++)

{

slots.Set(i, undef);

}

return slotArray;

}

Var\* JavascriptOperators::OP\_NewScopeSlotsWithoutPropIds(unsigned int count, int scopeIndex, ScriptContext \*scriptContext, FunctionBody \*functionBody)

{

DebuggerScope\* scope = Constants::FunctionBodyUnavailable;

if (scopeIndex != DebuggerScope::InvalidScopeIndex)

{

AssertMsg(functionBody->GetScopeObjectChain(), "A scope chain should always be created when there are new scope slots for blocks.");

scope = functionBody->GetScopeObjectChain()->pScopeChain->Item(scopeIndex);

}

return OP\_NewScopeSlots(count, scriptContext, scope);

}

Var\* JavascriptOperators::OP\_CloneScopeSlots(Var \*slotArray, ScriptContext \*scriptContext)

{

ScopeSlots slots(slotArray);

uint size = ScopeSlots::FirstSlotIndex + slots.GetCount();

Var\* slotArrayClone = RecyclerNewArray(scriptContext->GetRecycler(), Var, size);

memcpy\_s(slotArrayClone, sizeof(Var) \* size, slotArray, sizeof(Var) \* size);

return slotArrayClone;

}

Var JavascriptOperators::OP\_NewPseudoScope(ScriptContext \*scriptContext)

{

return scriptContext->GetLibrary()->CreatePseudoActivationObject();

}

Var JavascriptOperators::OP\_NewBlockScope(ScriptContext \*scriptContext)

{

return scriptContext->GetLibrary()->CreateBlockActivationObject();

}

Var JavascriptOperators::OP\_CloneBlockScope(BlockActivationObject \*blockScope, ScriptContext \*scriptContext)

{

return blockScope->Clone(scriptContext);

}

Var JavascriptOperators::OP\_IsInst(Var instance, Var aClass, ScriptContext\* scriptContext, IsInstInlineCache\* inlineCache)

{

if (!RecyclableObject::Is(aClass))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Operand\_Invalid\_NeedFunction, L"instanceof");

}

RecyclableObject\* constructor = RecyclableObject::FromVar(aClass);

if (scriptContext->GetConfig()->IsES6HasInstanceEnabled())

{

Var instOfHandler = JavascriptOperators::GetProperty(constructor, PropertyIds::\_symbolHasInstance, scriptContext);

if (JavascriptOperators::IsUndefinedObject(instOfHandler))

{

return JavascriptBoolean::ToVar(constructor->HasInstance(instance, scriptContext, inlineCache), scriptContext);

}

else

{

if (!JavascriptConversion::IsCallable(instOfHandler))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_NeedFunction, L"Symbol[Symbol.hasInstance]");

}

RecyclableObject \*instFunc = RecyclableObject::FromVar(instOfHandler);

Js::Var values[2];

Js::CallInfo info(Js::CallFlags\_Value, 2);

Js::Arguments args(info, values);

values[0] = constructor;

values[1] = instance;

Var result = JavascriptFunction::CallFunction<true>(instFunc, instFunc->GetEntryPoint(), args);

return JavascriptBoolean::ToVar(JavascriptConversion::ToBoolean(result, scriptContext) ? TRUE : FALSE, scriptContext);

}

}

else

{

return JavascriptBoolean::ToVar(constructor->HasInstance(instance, scriptContext, inlineCache), scriptContext);

}

}

void JavascriptOperators::OP\_InitClass(Var constructor, Var extends, ScriptContext \* scriptContext)

{

if (JavascriptOperators::GetTypeId(constructor) != Js::TypeId::TypeIds\_Function)

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Operand\_Invalid\_NeedFunction, L"class");

}

RecyclableObject \* ctor = RecyclableObject::FromVar(constructor);

// This is a circular reference to the constructor, it associate the constructor with the class and also allows us to check if a

// function is a constructor by comparing the homeObj to the this pointer. see ScriptFunction::IsClassConstructor() for implementation

JavascriptOperators::OP\_SetHomeObj(constructor, constructor);

if (extends)

{

switch (JavascriptOperators::GetTypeId(extends))

{

case Js::TypeId::TypeIds\_Null:

{

Var ctorProto = JavascriptOperators::GetProperty(constructor, ctor, Js::PropertyIds::prototype, scriptContext);

RecyclableObject \* ctorProtoObj = RecyclableObject::FromVar(ctorProto);

ctorProtoObj->SetPrototype(RecyclableObject::FromVar(extends));

ctorProtoObj->EnsureProperty(Js::PropertyIds::constructor);

ctorProtoObj->SetEnumerable(Js::PropertyIds::constructor, FALSE);

break;

}

default:

{

if (!RecyclableObject::Is(extends))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_InvalidPrototype, L"extends");

}

RecyclableObject \* extendsObj = RecyclableObject::FromVar(extends);

if (!JavascriptOperators::IsConstructor(extendsObj))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_ErrorOnNew);

}

if (!extendsObj->HasProperty(Js::PropertyIds::prototype))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_InvalidPrototype);

}

Var extendsProto = JavascriptOperators::GetProperty(extends, extendsObj, Js::PropertyIds::prototype, scriptContext);

uint extendsProtoTypeId = JavascriptOperators::GetTypeId(extendsProto);

if (extendsProtoTypeId <= Js::TypeId::TypeIds\_LastJavascriptPrimitiveType && extendsProtoTypeId != Js::TypeId::TypeIds\_Null)

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_InvalidPrototype);

}

Var ctorProto = JavascriptOperators::GetProperty(constructor, ctor, Js::PropertyIds::prototype, scriptContext);

RecyclableObject \* ctorProtoObj = RecyclableObject::FromVar(ctorProto);

ctorProtoObj->SetPrototype(RecyclableObject::FromVar(extendsProto));

ctorProtoObj->EnsureProperty(Js::PropertyIds::constructor);

ctorProtoObj->SetEnumerable(Js::PropertyIds::constructor, FALSE);

Var protoCtor = JavascriptOperators::GetProperty(ctorProto, ctorProtoObj, Js::PropertyIds::constructor, scriptContext);

RecyclableObject \* protoCtorObj = RecyclableObject::FromVar(protoCtor);

protoCtorObj->SetPrototype(extendsObj);

break;

}

}

}

}

void JavascriptOperators::OP\_LoadUndefinedToElement(Var instance, PropertyId propertyId)

{

AssertMsg(!TaggedNumber::Is(instance), "Invalid scope/root object");

JavascriptOperators::EnsureProperty(instance, propertyId);

}

void JavascriptOperators::OP\_LoadUndefinedToElementScoped(FrameDisplay \*pScope, PropertyId propertyId, Var defaultInstance, ScriptContext\* scriptContext)

{

int i;

int length = pScope->GetLength();

Var argInstance;

for (i = 0; i < length; i++)

{

argInstance = pScope->GetItem(i);

if (JavascriptOperators::EnsureProperty(argInstance, propertyId))

{

return;

}

}

if (!JavascriptOperators::HasOwnPropertyNoHostObject(defaultInstance, propertyId))

{

// CONSIDER : Consider adding pre-initialization support to activation objects.

JavascriptOperators::OP\_InitPropertyScoped(pScope, propertyId, scriptContext->GetLibrary()->GetUndefined(), defaultInstance, scriptContext);

}

}

void JavascriptOperators::OP\_LoadUndefinedToElementDynamic(Var instance, PropertyId propertyId, ScriptContext \*scriptContext)

{

if (!JavascriptOperators::HasOwnPropertyNoHostObject(instance, propertyId))

{

RecyclableObject::FromVar(instance)->InitPropertyScoped(propertyId, scriptContext->GetLibrary()->GetUndefined());

}

}

BOOL JavascriptOperators::EnsureProperty(Var instance, PropertyId propertyId)

{

RecyclableObject \*obj = RecyclableObject::FromVar(instance);

return (obj && obj->EnsureProperty(propertyId));

}

void JavascriptOperators::OP\_EnsureNoRootProperty(Var instance, PropertyId propertyId)

{

Assert(RootObjectBase::Is(instance));

RootObjectBase \*obj = RootObjectBase::FromVar(instance);

obj->EnsureNoProperty(propertyId);

}

void JavascriptOperators::OP\_EnsureNoRootRedeclProperty(Var instance, PropertyId propertyId)

{

Assert(RootObjectBase::Is(instance));

RecyclableObject \*obj = RecyclableObject::FromVar(instance);

obj->EnsureNoRedeclProperty(propertyId);

}

void JavascriptOperators::OP\_ScopedEnsureNoRedeclProperty(FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance)

{

int i;

int length = pDisplay->GetLength();

RecyclableObject \*object;

for (i = 0; i < length; i++)

{

object = RecyclableObject::FromVar(pDisplay->GetItem(i));

if (object->EnsureNoRedeclProperty(propertyId))

{

return;

}

}

object = RecyclableObject::FromVar(defaultInstance);

object->EnsureNoRedeclProperty(propertyId);

}

Var JavascriptOperators::IsIn(Var argProperty, Var instance, ScriptContext\* scriptContext)

{

// Note that the fact that we haven't seen a given name before doesn't mean that the instance doesn't

if (!IsObject(instance))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Operand\_Invalid\_NeedObject, L"in");

}

PropertyRecord const \* propertyRecord;

uint32 index;

IndexType indexType = GetIndexType(argProperty, scriptContext, &index, &propertyRecord, true);

RecyclableObject\* object = RecyclableObject::FromVar(instance);

BOOL result;

if( indexType == Js::IndexType\_Number )

{

result = JavascriptOperators::HasItem( object, index );

}

else

{

PropertyId propertyId = propertyRecord->GetPropertyId();

result = JavascriptOperators::HasProperty( object, propertyId );

#ifdef TELEMETRY\_JSO

{

Assert(indexType != Js::IndexType\_JavascriptString);

if( indexType == Js::IndexType\_PropertyId )

{

scriptContext->GetTelemetry().GetOpcodeTelemetry().IsIn( instance, propertyId, result != 0 );

}

}

#endif

}

return JavascriptBoolean::ToVar(result, scriptContext);

}

template <bool IsFromFullJit, class TInlineCache>

\_\_inline Var JavascriptOperators::PatchGetValue(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId)

{

return PatchGetValueWithThisPtr<IsFromFullJit, TInlineCache>(functionBody, inlineCache, inlineCacheIndex, instance, propertyId, instance);

}

template <bool IsFromFullJit, class TInlineCache>

\_\_forceinline Var JavascriptOperators::PatchGetValueWithThisPtr(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var thisInstance)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

Assert(Js::JavascriptStackWalker::ValidateTopJitFrame(scriptContext));

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined,

scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

else

{

return scriptContext->GetLibrary()->GetUndefined();

}

}

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

Var value;

if (CacheOperators::TryGetProperty<true, true, true, true, true, true, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

thisInstance, false, object, propertyId, &value, scriptContext, nullptr, &info))

{

return value;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchGetValue", propertyId, scriptContext, object);

}

#endif

return JavascriptOperators::GetProperty(thisInstance, object, propertyId, scriptContext, &info);

}

template Var JavascriptOperators::PatchGetValue<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchGetValue<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchGetValue<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchGetValue<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchGetValueWithThisPtr<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var thisInstance);

template Var JavascriptOperators::PatchGetValueWithThisPtr<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var thisInstance);

template Var JavascriptOperators::PatchGetValueWithThisPtr<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var thisInstance);

template Var JavascriptOperators::PatchGetValueWithThisPtr<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var thisInstance);

template <bool IsFromFullJit, class TInlineCache>

Var JavascriptOperators::PatchGetValueForTypeOf(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

Assert(Js::JavascriptStackWalker::ValidateTopJitFrame(scriptContext));

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined,

scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

else

{

return scriptContext->GetLibrary()->GetUndefined();

}

}

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

Var value;

if (CacheOperators::TryGetProperty<true, true, true, true, true, true, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

instance, false, object, propertyId, &value, scriptContext, nullptr, &info))

{

return value;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchGetValueForTypeOf", propertyId, scriptContext, object);

}

#endif

Var prop = nullptr;

BEGIN\_TYPEOF\_ERROR\_HANDLER(scriptContext);

prop = JavascriptOperators::GetProperty(instance, object, propertyId, scriptContext, &info);

END\_TYPEOF\_ERROR\_HANDLER(scriptContext, prop);

return prop;

}

template Var JavascriptOperators::PatchGetValueForTypeOf<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchGetValueForTypeOf<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchGetValueForTypeOf<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchGetValueForTypeOf<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

Var JavascriptOperators::PatchGetValueUsingSpecifiedInlineCache(InlineCache \* inlineCache, Var instance, RecyclableObject \* object, PropertyId propertyId, ScriptContext\* scriptContext)

{

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, inlineCache);

Var value;

if (CacheOperators::TryGetProperty<true, true, true, true, false, true, !InlineCache::IsPolymorphic, InlineCache::IsPolymorphic, false>(

instance, false, object, propertyId, &value, scriptContext, nullptr, &info))

{

return value;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchGetValue", propertyId, scriptContext, object);

}

#endif

return JavascriptOperators::GetProperty(instance, object, propertyId, scriptContext, &info);

}

Var JavascriptOperators::PatchGetValueNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId)

{

return PatchGetValueWithThisPtrNoFastPath(functionBody, inlineCache, inlineCacheIndex, instance, propertyId, instance);

}

Var JavascriptOperators::PatchGetValueWithThisPtrNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var thisInstance)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined,

scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

else

{

return scriptContext->GetLibrary()->GetUndefined();

}

}

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, true);

return JavascriptOperators::GetProperty(thisInstance, object, propertyId, scriptContext, &info);

}

template <bool IsFromFullJit, class TInlineCache>

\_\_inline Var JavascriptOperators::PatchGetRootValue(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject \* object, PropertyId propertyId)

{

AssertMsg(RootObjectBase::Is(object), "Root must be a global object!");

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

Var value;

if (CacheOperators::TryGetProperty<true, true, true, false, true, false, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, true, object, propertyId, &value, scriptContext, nullptr, &info))

{

return value;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchGetRootValue", propertyId, scriptContext, object);

}

#endif

return JavascriptOperators::OP\_GetRootProperty(object, propertyId, &info, scriptContext);

}

template Var JavascriptOperators::PatchGetRootValue<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject \* object, PropertyId propertyId);

template Var JavascriptOperators::PatchGetRootValue<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject \* object, PropertyId propertyId);

template Var JavascriptOperators::PatchGetRootValue<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject \* object, PropertyId propertyId);

template Var JavascriptOperators::PatchGetRootValue<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject \* object, PropertyId propertyId);

template <bool IsFromFullJit, class TInlineCache>

Var JavascriptOperators::PatchGetRootValueForTypeOf(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject \* object, PropertyId propertyId)

{

AssertMsg(RootObjectBase::Is(object), "Root must be a global object!");

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

Var value = nullptr;

if (CacheOperators::TryGetProperty<true, true, true, false, true, false, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, true, object, propertyId, &value, scriptContext, nullptr, &info))

{

return value;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchGetRootValueForTypeOf", propertyId, scriptContext, object);

}

#endif

value = nullptr;

BEGIN\_TYPEOF\_ERROR\_HANDLER(scriptContext);

if (JavascriptOperators::GetRootProperty(RecyclableObject::FromVar(object), propertyId, &value, scriptContext, &info))

{

if (scriptContext->IsUndeclBlockVar(value))

{

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UseBeforeDeclaration);

}

return value;

}

END\_TYPEOF\_ERROR\_HANDLER(scriptContext, value);

value = scriptContext->GetLibrary()->GetUndefined();

return value;

}

template Var JavascriptOperators::PatchGetRootValueForTypeOf<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject \* object, PropertyId propertyId);

template Var JavascriptOperators::PatchGetRootValueForTypeOf<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject \* object, PropertyId propertyId);

template Var JavascriptOperators::PatchGetRootValueForTypeOf<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject \* object, PropertyId propertyId);

template Var JavascriptOperators::PatchGetRootValueForTypeOf<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject \* object, PropertyId propertyId);

Var JavascriptOperators::PatchGetRootValueNoFastPath\_Var(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId)

{

return

PatchGetRootValueNoFastPath(

functionBody,

inlineCache,

inlineCacheIndex,

DynamicObject::FromVar(instance),

propertyId);

}

Var JavascriptOperators::PatchGetRootValueNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId)

{

AssertMsg(RootObjectBase::Is(object), "Root must be a global object!");

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, true);

return JavascriptOperators::OP\_GetRootProperty(object, propertyId, &info, scriptContext);

}

template <bool IsFromFullJit, class TInlineCache>

\_\_inline Var JavascriptOperators::PatchGetPropertyScoped(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance)

{

// Get the property, using a scope stack rather than an individual instance.

// Walk the stack until we find an instance that has the property.

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

uint16 length = pDisplay->GetLength();

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

for (uint16 i = 0; i < length; i++)

{

DynamicObject\* object = (DynamicObject\*)pDisplay->GetItem(i);

Var value;

if (CacheOperators::TryGetProperty<true, true, true, false, true, true, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, false, object, propertyId, &value, scriptContext, nullptr, &info))

{

return value;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchGetPropertyScoped", propertyId, scriptContext, object);

}

#endif

if (JavascriptOperators::GetProperty(object, propertyId, &value, scriptContext, &info))

{

if (scriptContext->IsUndeclBlockVar(value) && propertyId != PropertyIds::\_lexicalThisSlotSymbol)

{

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UseBeforeDeclaration);

}

return value;

}

}

// No one in the scope stack has the property, so get it from the default instance provided by the caller.

Var value = JavascriptOperators::PatchGetRootValue<IsFromFullJit>(functionBody, inlineCache, inlineCacheIndex, DynamicObject::FromVar(defaultInstance), propertyId);

if (scriptContext->IsUndeclBlockVar(value))

{

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UseBeforeDeclaration);

}

return value;

}

template Var JavascriptOperators::PatchGetPropertyScoped<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance);

template Var JavascriptOperators::PatchGetPropertyScoped<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance);

template Var JavascriptOperators::PatchGetPropertyScoped<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance);

template Var JavascriptOperators::PatchGetPropertyScoped<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance);

template <bool IsFromFullJit, class TInlineCache>

Var JavascriptOperators::PatchGetPropertyForTypeOfScoped(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance)

{

Var value = nullptr;

ScriptContext \*scriptContext = functionBody->GetScriptContext();

BEGIN\_TYPEOF\_ERROR\_HANDLER(scriptContext);

value = JavascriptOperators::PatchGetPropertyScoped<IsFromFullJit, TInlineCache>(functionBody, inlineCache, inlineCacheIndex, pDisplay, propertyId, defaultInstance);

END\_TYPEOF\_ERROR\_HANDLER(scriptContext, value)

return value;

}

template Var JavascriptOperators::PatchGetPropertyForTypeOfScoped<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance);

template Var JavascriptOperators::PatchGetPropertyForTypeOfScoped<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance);

template Var JavascriptOperators::PatchGetPropertyForTypeOfScoped<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance);

template Var JavascriptOperators::PatchGetPropertyForTypeOfScoped<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance);

template <bool IsFromFullJit, class TInlineCache>

\_\_inline Var JavascriptOperators::PatchGetMethod(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId)

{

Assert(inlineCache != nullptr);

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

RecyclableObject\* object = nullptr;

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

// Don't error if we disabled implicit calls

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined,

scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

else

{

#ifdef TELEMETRY\_JSO

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetMethod throws an exception if not found.

scriptContext->GetTelemetry().GetOpcodeTelemetry().GetMethodProperty(object, propertyId, value, /\*successful:\*/false);

}

#endif

return scriptContext->GetLibrary()->GetUndefined();

}

}

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

Var value;

if (CacheOperators::TryGetProperty<true, true, true, false, true, true, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

instance, false, object, propertyId, &value, scriptContext, nullptr, &info))

{

return value;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchGetMethod", propertyId, scriptContext, object);

}

#endif

value = Js::JavascriptOperators::PatchGetMethodFromObject(instance, object, propertyId, &info, scriptContext, false);

#ifdef TELEMETRY\_JSO

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetMethod throws an exception if not found.

scriptContext->GetTelemetry().GetOpcodeTelemetry().GetMethodProperty(object, propertyId, value, /\*successful:\*/true);

}

#endif

return value;

}

template Var JavascriptOperators::PatchGetMethod<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchGetMethod<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchGetMethod<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchGetMethod<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template <bool IsFromFullJit, class TInlineCache>

\_\_inline Var JavascriptOperators::PatchGetRootMethod(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId)

{

Assert(inlineCache != nullptr);

AssertMsg(RootObjectBase::Is(object), "Root must be a global object!");

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

Var value;

if (CacheOperators::TryGetProperty<true, true, true, false, true, false, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, true, object, propertyId, &value, scriptContext, nullptr, &info))

{

return value;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchGetRootMethod", propertyId, scriptContext, object);

}

#endif

value = Js::JavascriptOperators::PatchGetMethodFromObject(object, object, propertyId, &info, scriptContext, true);

#ifdef TELEMETRY\_JSO

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetMethod throws an exception if not found.

scriptContext->GetTelemetry().GetOpcodeTelemetry().GetMethodProperty(object, propertyId, value, /\*successful:\*/ true);

}

#endif

return value;

}

template Var JavascriptOperators::PatchGetRootMethod<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId);

template Var JavascriptOperators::PatchGetRootMethod<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId);

template Var JavascriptOperators::PatchGetRootMethod<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId);

template Var JavascriptOperators::PatchGetRootMethod<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId);

template <bool IsFromFullJit, class TInlineCache>

\_\_inline Var JavascriptOperators::PatchScopedGetMethod(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId)

{

Assert(inlineCache != nullptr);

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

// Don't error if we disabled implicit calls

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined,

scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

else

{

return scriptContext->GetLibrary()->GetUndefined();

}

}

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

const bool isRoot = RootObjectBase::Is(object);

Var value;

if (CacheOperators::TryGetProperty<true, true, true, false, true, false, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

instance, isRoot, object, propertyId, &value, scriptContext, nullptr, &info))

{

return value;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchGetMethod", propertyId, scriptContext, object);

}

#endif

return Js::JavascriptOperators::PatchGetMethodFromObject(instance, object, propertyId, &info, scriptContext, isRoot);

}

template Var JavascriptOperators::PatchScopedGetMethod<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchScopedGetMethod<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchScopedGetMethod<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchScopedGetMethod<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

Var JavascriptOperators::PatchGetMethodNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

// Don't error if we disabled implicit calls

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined,

scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

else

{

return scriptContext->GetLibrary()->GetUndefined();

}

}

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, true);

return Js::JavascriptOperators::PatchGetMethodFromObject(instance, object, propertyId, &info, scriptContext, false);

}

Var JavascriptOperators::PatchGetRootMethodNoFastPath\_Var(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId)

{

return

PatchGetRootMethodNoFastPath(

functionBody,

inlineCache,

inlineCacheIndex,

DynamicObject::FromVar(instance),

propertyId);

}

Var JavascriptOperators::PatchGetRootMethodNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId)

{

AssertMsg(RootObjectBase::Is(object), "Root must be a global object!");

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, true);

return Js::JavascriptOperators::PatchGetMethodFromObject(object, object, propertyId, &info, functionBody->GetScriptContext(), true);

}

Var JavascriptOperators::PatchGetMethodFromObject(Var instance, RecyclableObject\* propertyObject, PropertyId propertyId, PropertyValueInfo \* info, ScriptContext\* scriptContext, bool isRootLd)

{

Assert(IsPropertyObject(propertyObject));

Var value = nullptr;

BOOL foundValue = FALSE;

if (isRootLd)

{

RootObjectBase\* rootObject = RootObjectBase::FromVar(instance);

foundValue = JavascriptOperators::GetRootPropertyReference(rootObject, propertyId, &value, scriptContext, info);

}

else

{

foundValue = JavascriptOperators::GetPropertyReference(instance, propertyObject, propertyId, &value, scriptContext, info);

}

if (!foundValue)

{

// Don't error if we disabled implicit calls

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

const wchar\_t\* propertyName = scriptContext->GetPropertyName(propertyId)->GetBuffer();

value = scriptContext->GetLibrary()->GetUndefined();

JavascriptFunction \* caller = NULL;

if (JavascriptStackWalker::GetCaller(&caller, scriptContext))

{

FunctionBody \* callerBody = caller->GetFunctionBody();

if (callerBody && callerBody->GetUtf8SourceInfo()->GetIsXDomain())

{

propertyName = NULL;

}

}

// Prior to version 12 we had mistakenly immediately thrown an error for property reference method calls

// (i.e. <expr>.foo() form) when the target object is the global object. The spec says that a GetValue

// on a reference should throw if the reference is unresolved, of which a property reference can never be,

// however it can be unresolved in the case of an identifier expression, e.g. foo() with no qualification.

// Such a case would come down to the global object if foo was undefined, hence the check for root object,

// except that it should have been a check for isRootLd to be correct.

//

// // (at global scope)

// foo(x());

//

// should throw an error before evaluating x() if foo is not defined, but

//

// // (at global scope)

// this.foo(x());

//

// should evaluate x() before throwing an error if foo is not a property on the global object.

// Maintain old behavior prior to version 12.

bool isPropertyReference = !isRootLd;

if (!isPropertyReference)

{

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UndefVariable, propertyName);

}

else

{

// ES5 11.2.3 #2: We evaluate the call target but don't throw yet if target member is missing. We need to evaluate argList

// first (#3). Postpone throwing error to invoke time.

value = ThrowErrorObject::CreateThrowTypeErrorObject(scriptContext, VBSERR\_OLENoPropOrMethod, propertyName);

}

}

}

return value;

}

template <bool IsFromFullJit, class TInlineCache>

\_\_inline void JavascriptOperators::PatchPutValue(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags)

{

return PatchPutValueWithThisPtr<IsFromFullJit, TInlineCache>(functionBody, inlineCache, inlineCacheIndex, instance, propertyId, newValue, instance, flags);

}

template <bool IsFromFullJit, class TInlineCache>

\_\_inline void JavascriptOperators::PatchPutValueWithThisPtr(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, Var thisInstance, PropertyOperationFlags flags)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

if (TaggedNumber::Is(instance))

{

JavascriptOperators::SetPropertyOnTaggedNumber(instance, nullptr, propertyId, newValue, scriptContext, flags);

return;

}

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

RecyclableObject\* object = RecyclableObject::FromVar(instance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

if (CacheOperators::TrySetProperty<true, true, true, true, true, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, false, propertyId, newValue, scriptContext, flags, nullptr, &info))

{

return;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchPutValue", propertyId, scriptContext, object);

}

#endif

ImplicitCallFlags prevImplicitCallFlags = ImplicitCall\_None;

ImplicitCallFlags currImplicitCallFlags = ImplicitCall\_None;

bool hasThisOnlyStatements = functionBody->GetHasOnlyThisStmts();

if (hasThisOnlyStatements)

{

prevImplicitCallFlags = CacheAndClearImplicitBit(scriptContext);

}

if (!JavascriptOperators::OP\_SetProperty(object, propertyId, newValue, scriptContext, &info, flags, thisInstance))

{

// Add implicit call flags, to bail out if field copy prop may propagate the wrong value.

scriptContext->GetThreadContext()->AddImplicitCallFlags(ImplicitCall\_NoOpSet);

}

if (hasThisOnlyStatements)

{

currImplicitCallFlags = CheckAndUpdateFunctionBodyWithImplicitFlag(functionBody);

RestoreImplicitFlag(scriptContext, prevImplicitCallFlags, currImplicitCallFlags);

}

}

template void JavascriptOperators::PatchPutValue<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutValue<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutValue<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutValue<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template <bool IsFromFullJit, class TInlineCache>

\_\_inline void JavascriptOperators::PatchPutRootValue(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

RecyclableObject\* object = RecyclableObject::FromVar(instance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

if (CacheOperators::TrySetProperty<true, true, true, true, false, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, true, propertyId, newValue, scriptContext, flags, nullptr, &info))

{

return;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchPutRootValue", propertyId, scriptContext, object);

}

#endif

ImplicitCallFlags prevImplicitCallFlags = ImplicitCall\_None;

ImplicitCallFlags currImplicitCallFlags = ImplicitCall\_None;

bool hasThisOnlyStatements = functionBody->GetHasOnlyThisStmts();

if (hasThisOnlyStatements)

{

prevImplicitCallFlags = CacheAndClearImplicitBit(scriptContext);

}

if (!JavascriptOperators::SetRootProperty(object, propertyId, newValue, &info, scriptContext, flags))

{

// Add implicit call flags, to bail out if field copy prop may propagate the wrong value.

scriptContext->GetThreadContext()->AddImplicitCallFlags(ImplicitCall\_NoOpSet);

}

if (hasThisOnlyStatements)

{

currImplicitCallFlags = CheckAndUpdateFunctionBodyWithImplicitFlag(functionBody);

RestoreImplicitFlag(scriptContext, prevImplicitCallFlags, currImplicitCallFlags);

}

}

template void JavascriptOperators::PatchPutRootValue<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutRootValue<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutRootValue<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutRootValue<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template <bool IsFromFullJit, class TInlineCache>

\_\_inline void JavascriptOperators::PatchPutValueNoLocalFastPath(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

if (TaggedNumber::Is(instance))

{

JavascriptOperators::SetPropertyOnTaggedNumber(instance,

nullptr,

propertyId,

newValue,

scriptContext,

flags);

return;

}

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

RecyclableObject \*object = RecyclableObject::FromVar(instance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

if (CacheOperators::TrySetProperty<!TInlineCache::IsPolymorphic, true, true, true, true, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, false, propertyId, newValue, scriptContext, flags, nullptr, &info))

{

return;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchPutValueNoLocalFastPath", propertyId, scriptContext, object);

}

#endif

ImplicitCallFlags prevImplicitCallFlags = ImplicitCall\_None;

ImplicitCallFlags currImplicitCallFlags = ImplicitCall\_None;

bool hasThisOnlyStatements = functionBody->GetHasOnlyThisStmts();

if (hasThisOnlyStatements)

{

prevImplicitCallFlags = CacheAndClearImplicitBit(scriptContext);

}

if (!JavascriptOperators::OP\_SetProperty(instance, propertyId, newValue, scriptContext, &info, flags))

{

// Add implicit call flags, to bail out if field copy prop may propagate the wrong value.

scriptContext->GetThreadContext()->AddImplicitCallFlags(ImplicitCall\_NoOpSet);

}

if (hasThisOnlyStatements)

{

currImplicitCallFlags = CheckAndUpdateFunctionBodyWithImplicitFlag(functionBody);

RestoreImplicitFlag(scriptContext, prevImplicitCallFlags, currImplicitCallFlags);

}

}

template void JavascriptOperators::PatchPutValueNoLocalFastPath<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutValueNoLocalFastPath<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutValueNoLocalFastPath<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutValueNoLocalFastPath<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template <bool IsFromFullJit, class TInlineCache>

\_\_inline void JavascriptOperators::PatchPutValueWithThisPtrNoLocalFastPath(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, Var thisInstance, PropertyOperationFlags flags)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

if (TaggedNumber::Is(instance))

{

JavascriptOperators::SetPropertyOnTaggedNumber(instance,

nullptr,

propertyId,

newValue,

scriptContext,

flags);

return;

}

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

RecyclableObject \*object = RecyclableObject::FromVar(instance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

if (CacheOperators::TrySetProperty<!TInlineCache::IsPolymorphic, true, true, true, true, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, false, propertyId, newValue, scriptContext, flags, nullptr, &info))

{

return;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchPutValueNoLocalFastPath", propertyId, scriptContext, object);

}

#endif

ImplicitCallFlags prevImplicitCallFlags = ImplicitCall\_None;

ImplicitCallFlags currImplicitCallFlags = ImplicitCall\_None;

bool hasThisOnlyStatements = functionBody->GetHasOnlyThisStmts();

if (hasThisOnlyStatements)

{

prevImplicitCallFlags = CacheAndClearImplicitBit(scriptContext);

}

if (!JavascriptOperators::OP\_SetProperty(instance, propertyId, newValue, scriptContext, &info, flags, thisInstance))

{

// Add implicit call flags, to bail out if field copy prop may propagate the wrong value.

scriptContext->GetThreadContext()->AddImplicitCallFlags(ImplicitCall\_NoOpSet);

}

if (hasThisOnlyStatements)

{

currImplicitCallFlags = CheckAndUpdateFunctionBodyWithImplicitFlag(functionBody);

RestoreImplicitFlag(scriptContext, prevImplicitCallFlags, currImplicitCallFlags);

}

}

template void JavascriptOperators::PatchPutValueWithThisPtrNoLocalFastPath<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, Var thisInstance, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutValueWithThisPtrNoLocalFastPath<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, Var thisInstance, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutValueWithThisPtrNoLocalFastPath<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, Var thisInstance, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutValueWithThisPtrNoLocalFastPath<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, Var thisInstance, PropertyOperationFlags flags);

template <bool IsFromFullJit, class TInlineCache>

\_\_inline void JavascriptOperators::PatchPutRootValueNoLocalFastPath(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

RecyclableObject \*object = RecyclableObject::FromVar(instance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

if (CacheOperators::TrySetProperty<!TInlineCache::IsPolymorphic, true, true, true, false, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, true, propertyId, newValue, scriptContext, flags, nullptr, &info))

{

return;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchPutRootValueNoLocalFastPath", propertyId, scriptContext, object);

}

#endif

ImplicitCallFlags prevImplicitCallFlags = ImplicitCall\_None;

ImplicitCallFlags currImplicitCallFlags = ImplicitCall\_None;

bool hasThisOnlyStatements = functionBody->GetHasOnlyThisStmts();

if (hasThisOnlyStatements)

{

prevImplicitCallFlags = CacheAndClearImplicitBit(scriptContext);

}

if (!JavascriptOperators::SetRootProperty(object, propertyId, newValue, &info, scriptContext, flags))

{

// Add implicit call flags, to bail out if field copy prop may propagate the wrong value.

scriptContext->GetThreadContext()->AddImplicitCallFlags(ImplicitCall\_NoOpSet);

}

if (hasThisOnlyStatements)

{

currImplicitCallFlags = CheckAndUpdateFunctionBodyWithImplicitFlag(functionBody);

RestoreImplicitFlag(scriptContext, prevImplicitCallFlags, currImplicitCallFlags);

}

}

template void JavascriptOperators::PatchPutRootValueNoLocalFastPath<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutRootValueNoLocalFastPath<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutRootValueNoLocalFastPath<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutRootValueNoLocalFastPath<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

void JavascriptOperators::PatchPutValueNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags)

{

PatchPutValueWithThisPtrNoFastPath(functionBody, inlineCache, inlineCacheIndex, instance, propertyId, newValue, instance, flags);

}

void JavascriptOperators::PatchPutValueWithThisPtrNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, Var thisInstance, PropertyOperationFlags flags)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

if (TaggedNumber::Is(instance))

{

JavascriptOperators::SetPropertyOnTaggedNumber(instance, nullptr, propertyId, newValue, scriptContext, flags);

return;

}

RecyclableObject\* object = RecyclableObject::FromVar(instance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, true);

if (!JavascriptOperators::OP\_SetProperty(object, propertyId, newValue, scriptContext, &info, flags, thisInstance))

{

// Add implicit call flags, to bail out if field copy prop may propagate the wrong value.

scriptContext->GetThreadContext()->AddImplicitCallFlags(ImplicitCall\_NoOpSet);

}

}

void JavascriptOperators::PatchPutRootValueNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

RecyclableObject\* object = RecyclableObject::FromVar(instance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, true);

if (!JavascriptOperators::SetRootProperty(object, propertyId, newValue, &info, scriptContext, flags))

{

// Add implicit call flags, to bail out if field copy prop may propagate the wrong value.

scriptContext->GetThreadContext()->AddImplicitCallFlags(ImplicitCall\_NoOpSet);

}

}

template <bool IsFromFullJit, class TInlineCache>

\_\_inline void JavascriptOperators::PatchInitValue(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, RecyclableObject\* object, PropertyId propertyId, Var newValue)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

const PropertyOperationFlags flags = newValue == NULL ? PropertyOperation\_SpecialValue : PropertyOperation\_None;

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

if (CacheOperators::TrySetProperty<true, true, false, true, true, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, false, propertyId, newValue, scriptContext, flags, nullptr, &info))

{

return;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchInitValue", propertyId, scriptContext, object);

}

#endif

Type \*typeWithoutProperty = object->GetType();

// Ideally the lowerer would emit a call to the right flavor of PatchInitValue, so that we can ensure that we only

// ever initialize to NULL in the right cases. But the backend uses the StFld opcode for initialization, and it

// would be cumbersome to thread the different helper calls all the way down

if (object->InitProperty(propertyId, newValue, flags, &info))

{

CacheOperators::CachePropertyWrite(object, false, typeWithoutProperty, propertyId, &info, scriptContext);

}

}

template void JavascriptOperators::PatchInitValue<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, RecyclableObject\* object, PropertyId propertyId, Var newValue);

template void JavascriptOperators::PatchInitValue<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, RecyclableObject\* object, PropertyId propertyId, Var newValue);

template void JavascriptOperators::PatchInitValue<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, RecyclableObject\* object, PropertyId propertyId, Var newValue);

template void JavascriptOperators::PatchInitValue<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, RecyclableObject\* object, PropertyId propertyId, Var newValue);

void JavascriptOperators::PatchInitValueNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, RecyclableObject\* object, PropertyId propertyId, Var newValue)

{

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, true);

Type \*typeWithoutProperty = object->GetType();

if (object->InitProperty(propertyId, newValue, PropertyOperation\_None, &info))

{

CacheOperators::CachePropertyWrite(object, false, typeWithoutProperty, propertyId, &info, functionBody->GetScriptContext());

}

}

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

void JavascriptOperators::TracePropertyEquivalenceCheck(const JitEquivalentTypeGuard\* guard, const Type\* type, const Type\* refType, bool isEquivalent, uint failedPropertyIndex)

{

if (PHASE\_TRACE1(Js::EquivObjTypeSpecPhase))

{

uint propertyCount = guard->GetCache()->record.propertyCount;

Output::Print(L"EquivObjTypeSpec: checking %u properties on operation %u, (type = 0x%p, ref type = 0x%p):\n",

propertyCount, guard->GetObjTypeSpecFldId(), type, refType);

const Js::TypeEquivalenceRecord& record = guard->GetCache()->record;

ScriptContext\* scriptContext = type->GetScriptContext();

if (isEquivalent)

{

if (Js::Configuration::Global.flags.Verbose)

{

Output::Print(L" <start>, ");

for (uint pi = 0; pi < propertyCount; pi++)

{

const EquivalentPropertyEntry\* refInfo = &record.properties[pi];

const PropertyRecord\* propertyRecord = scriptContext->GetPropertyName(refInfo->propertyId);

Output::Print(L"%s(#%d)@%ua%dw%d, ", propertyRecord->GetBuffer(), propertyRecord->GetPropertyId(), refInfo->slotIndex, refInfo->isAuxSlot, refInfo->mustBeWritable);

}

Output::Print(L"<end>\n");

}

}

else

{

const EquivalentPropertyEntry\* refInfo = &record.properties[failedPropertyIndex];

Js::PropertyEquivalenceInfo info(Constants::NoSlot, false, false);

const PropertyRecord\* propertyRecord = scriptContext->GetPropertyName(refInfo->propertyId);

if (DynamicType::Is(type->GetTypeId()))

{

Js::DynamicTypeHandler\* typeHandler = (static\_cast<const DynamicType\*>(type))->GetTypeHandler();

typeHandler->GetPropertyEquivalenceInfo(propertyRecord, info);

}

Output::Print(L"EquivObjTypeSpec: check failed for %s (#%d) on operation %u:\n",

propertyRecord->GetBuffer(), propertyRecord->GetPropertyId(), guard->GetObjTypeSpecFldId());

Output::Print(L" type = 0x%p, ref type = 0x%p, slot = 0x%u (%d), ref slot = 0x%u (%d), is writable = %d, required writable = %d\n",

type, refType, info.slotIndex, refInfo->slotIndex, info.isAuxSlot, refInfo->isAuxSlot, info.isWritable, refInfo->mustBeWritable);

}

Output::Flush();

}

}

#endif

bool JavascriptOperators::IsStaticTypeObjTypeSpecEquivalent(const TypeEquivalenceRecord& equivalenceRecord, uint& failedIndex)

{

uint propertyCount = equivalenceRecord.propertyCount;

Js::EquivalentPropertyEntry\* properties = equivalenceRecord.properties;

for (uint pi = 0; pi < propertyCount; pi++)

{

const EquivalentPropertyEntry\* refInfo = &properties[pi];

if (!IsStaticTypeObjTypeSpecEquivalent(refInfo))

{

failedIndex = pi;

return false;

}

}

return true;

}

bool JavascriptOperators::IsStaticTypeObjTypeSpecEquivalent(const EquivalentPropertyEntry \*entry)

{

// Objects of static types have no local properties, but they may load fields from their prototypes.

return entry->slotIndex == Constants::NoSlot && !entry->mustBeWritable;

}

bool JavascriptOperators::CheckIfTypeIsEquivalent(Type\* type, JitEquivalentTypeGuard\* guard)

{

if (guard->GetValue() == 0)

{

return false;

}

if (guard->GetType()->GetScriptContext() != type->GetScriptContext())

{

// Can't cache cross-context objects

return false;

}

// CONSIDER : Add stats on how often the cache hits, and simply force bailout if

// the efficacy is too low.

EquivalentTypeCache\* cache = guard->GetCache();

// CONSIDER : Consider emitting o.type == equivTypes[hash(o.type)] in machine code before calling

// this helper, particularly if we want to handle polymorphism with frequently changing types.

Assert(EQUIVALENT\_TYPE\_CACHE\_SIZE == 8);

Type\*\* equivTypes = cache->types;

if (type == equivTypes[0] || type == equivTypes[1] || type == equivTypes[2] || type == equivTypes[3] ||

type == equivTypes[4] || type == equivTypes[5] || type == equivTypes[6] || type == equivTypes[7])

{

guard->SetType(type);

return true;

}

// If we didn't find the type in the cache, let's check if it's equivalent the slow way, by comparing

// each of its relevant property slots to its equivalent in one of the cached types.

// We are making a few assumption that simplify the process:

// 1. If two types have the same prototype, any properties loaded from a prototype must come from the same slot.

// If any of the prototypes in the chain was altered such that this is no longer true, the corresponding

// property guard would have been invalidated and we would bail out at the guard check (either on this

// type check or downstream, but before the property load is attempted).

// 2. For polymorphic field loads fixed fields are only supported on prototypes. Hence, if two types have the

// same prototype, any of the equivalent fixed properties will match. If any has been overwritten, the

// corresponding guard would have been invalidated and we would bail out (as above).

Type\* refType = equivTypes[0];

if (refType == nullptr)

{

return false;

}

if (cache->IsLoadedFromProto() && type->GetPrototype() != refType->GetPrototype())

{

if (PHASE\_TRACE1(Js::EquivObjTypeSpecPhase))

{

Output::Print(L"EquivObjTypeSpec: failed check on operation %u (type = 0x%x, ref type = 0x%x, proto = 0x%x, ref proto = 0x%x) \n",

guard->GetObjTypeSpecFldId(), type, refType, type->GetPrototype(), refType->GetPrototype());

Output::Flush();

}

return false;

}

if (type->GetTypeId() != refType->GetTypeId())

{

if (PHASE\_TRACE1(Js::EquivObjTypeSpecPhase))

{

Output::Print(L"EquivObjTypeSpec: failed check on operation %u (type = 0x%x, ref type = 0x%x, proto = 0x%x, ref proto = 0x%x) \n",

guard->GetObjTypeSpecFldId(), type, refType, type->GetPrototype(), refType->GetPrototype());

Output::Flush();

}

return false;

}

// Review : This is quite slow. We could make it somewhat faster, by keeping slot indexes instead

// of property IDs, but that would mean we would need to look up property IDs from slot indexes when installing

// property guards, or maintain a whole separate list of equivalent slot indexes.

Assert(cache->record.propertyCount > 0);

// CONSIDER (EquivObjTypeSpec): Impose a limit on the number of properties guarded by an equivalent type check.

// The trick is where in the glob opt to make the cut off. Perhaps in the forward pass we could track the number of

// field operations protected by a type check (keep a counter on the type's value info), and if that counter exceeds

// some threshold, simply stop optimizing any further instructions.

bool isEquivalent;

uint failedPropertyIndex;

if (DynamicType::Is(type->GetTypeId()))

{

Js::DynamicTypeHandler\* typeHandler = (static\_cast<DynamicType\*>(type))->GetTypeHandler();

isEquivalent = typeHandler->IsObjTypeSpecEquivalent(type, cache->record, failedPropertyIndex);

}

else

{

Assert(StaticType::Is(type->GetTypeId()));

isEquivalent = IsStaticTypeObjTypeSpecEquivalent(cache->record, failedPropertyIndex);

}

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

TracePropertyEquivalenceCheck(guard, type, refType, isEquivalent, failedPropertyIndex);

#endif

if (!isEquivalent)

{

return false;

}

// CONSIDER (EquivObjTypeSpec): Invent some form of least recently used eviction scheme.

uintptr\_t index = (reinterpret\_cast<uintptr\_t>(type) >> 4) & (EQUIVALENT\_TYPE\_CACHE\_SIZE - 1);

if (cache->nextEvictionVictim == EQUIVALENT\_TYPE\_CACHE\_SIZE)

{

\_\_analysis\_assume(index < EQUIVALENT\_TYPE\_CACHE\_SIZE);

if (equivTypes[index] != nullptr)

{

uintptr\_t initialIndex = index;

index = (initialIndex + 1) & (EQUIVALENT\_TYPE\_CACHE\_SIZE - 1);

for (; index != initialIndex; index = (index + 1) & (EQUIVALENT\_TYPE\_CACHE\_SIZE - 1))

{

if (equivTypes[index] == nullptr) break;

}

}

\_\_analysis\_assume(index < EQUIVALENT\_TYPE\_CACHE\_SIZE);

if (equivTypes[index] != nullptr)

{

cache->nextEvictionVictim = 0;

}

}

else

{

Assert(cache->nextEvictionVictim < EQUIVALENT\_TYPE\_CACHE\_SIZE);

\_\_analysis\_assume(cache->nextEvictionVictim < EQUIVALENT\_TYPE\_CACHE\_SIZE);

equivTypes[cache->nextEvictionVictim] = equivTypes[index];

cache->nextEvictionVictim = (cache->nextEvictionVictim + 1) & (EQUIVALENT\_TYPE\_CACHE\_SIZE - 1);

}

Assert(index < EQUIVALENT\_TYPE\_CACHE\_SIZE);

\_\_analysis\_assume(index < EQUIVALENT\_TYPE\_CACHE\_SIZE);

equivTypes[index] = type;

if (cache->HasFixedValue())

{

// Fixed field checks allow us to assume a specific type ID, but the assumption is only

// valid if we lock the type. Otherwise, the type ID may change out from under us without

// evolving the type.

if (DynamicType::Is(type->GetTypeId()))

{

DynamicType \*dynamicType = static\_cast<DynamicType\*>(type);

if (!dynamicType->GetIsLocked())

{

dynamicType->LockType();

}

}

}

guard->SetType(type);

return true;

}

void JavascriptOperators::GetPropertyIdForInt(uint64 value, ScriptContext\* scriptContext, PropertyRecord const \*\* propertyRecord)

{

wchar\_t buffer[20];

::\_ui64tow\_s(value, buffer, sizeof(buffer)/sizeof(wchar\_t), 10);

scriptContext->GetOrAddPropertyRecord(buffer, JavascriptString::GetBufferLength(buffer), propertyRecord);

}

void JavascriptOperators::GetPropertyIdForInt(uint32 value, ScriptContext\* scriptContext, PropertyRecord const \*\* propertyRecord)

{

GetPropertyIdForInt(static\_cast<uint64>(value), scriptContext, propertyRecord);

}

Var JavascriptOperators::FromPropertyDescriptor(PropertyDescriptor descriptor, ScriptContext\* scriptContext)

{

DynamicObject\* object = scriptContext->GetLibrary()->CreateObject();

// ES5 Section 8.10.4 specifies the order for adding these properties.

if (descriptor.IsDataDescriptor())

{

if (descriptor.ValueSpecified())

{

JavascriptOperators::InitProperty(object, PropertyIds::value, descriptor.GetValue());

}

JavascriptOperators::InitProperty(object, PropertyIds::writable, JavascriptBoolean::ToVar(descriptor.IsWritable(),scriptContext));

}

else if (descriptor.IsAccessorDescriptor())

{

JavascriptOperators::InitProperty(object, PropertyIds::get, JavascriptOperators::CanonicalizeAccessor(descriptor.GetGetter(), scriptContext));

JavascriptOperators::InitProperty(object, PropertyIds::set, JavascriptOperators::CanonicalizeAccessor(descriptor.GetSetter(), scriptContext));

}

if (descriptor.EnumerableSpecified())

{

JavascriptOperators::InitProperty(object, PropertyIds::enumerable, JavascriptBoolean::ToVar(descriptor.IsEnumerable(), scriptContext));

}

if (descriptor.ConfigurableSpecified())

{

JavascriptOperators::InitProperty(object, PropertyIds::configurable, JavascriptBoolean::ToVar(descriptor.IsConfigurable(), scriptContext));

}

return object;

}

// ES5 8.12.9 [[DefineOwnProperty]].

// Return value:

// - TRUE = success.

// - FALSE (can throw depending on throwOnError parameter) = unsuccessful.

BOOL JavascriptOperators::DefineOwnPropertyDescriptor(RecyclableObject\* obj, PropertyId propId, const PropertyDescriptor& descriptor, bool throwOnError, ScriptContext\* scriptContext)

{

Assert(obj);

Assert(scriptContext);

if (JavascriptProxy::Is(obj))

{

return JavascriptProxy::DefineOwnPropertyDescriptor(obj, propId, descriptor, throwOnError, scriptContext);

}

PropertyDescriptor currentDescriptor;

BOOL isCurrentDescriptorDefined = JavascriptOperators::GetOwnPropertyDescriptor(obj, propId, scriptContext, &currentDescriptor);

bool isExtensible = !!obj->IsExtensible();

return ValidateAndApplyPropertyDescriptor<true>(obj, propId, descriptor, isCurrentDescriptorDefined ? &currentDescriptor : nullptr, isExtensible, throwOnError, scriptContext);

}

BOOL JavascriptOperators::IsCompatiblePropertyDescriptor(const PropertyDescriptor& descriptor, PropertyDescriptor\* currentDescriptor, bool isExtensible, bool throwOnError, ScriptContext\* scriptContext)

{

return ValidateAndApplyPropertyDescriptor<false>(nullptr, Constants::NoProperty, descriptor, currentDescriptor, isExtensible, throwOnError, scriptContext);

}

template<bool needToSetProperty>

BOOL JavascriptOperators::ValidateAndApplyPropertyDescriptor(RecyclableObject\* obj, PropertyId propId, const PropertyDescriptor& descriptor,

PropertyDescriptor\* currentDescriptor, bool isExtensible, bool throwOnError, ScriptContext\* scriptContext)

{

Var defaultDataValue = scriptContext->GetLibrary()->GetUndefined();

Var defaultAccessorValue = scriptContext->GetLibrary()->GetDefaultAccessorFunction();

if (currentDescriptor == nullptr)

{

if (!isExtensible) // ES5 8.12.9.3.

{

return Reject(throwOnError, scriptContext, JSERR\_DefineProperty\_NotExtensible, propId);

}

else // ES5 8.12.9.4.

{

if (needToSetProperty)

{

if (descriptor.IsGenericDescriptor() || descriptor.IsDataDescriptor())

{

// ES5 8.12.9.4a: Create an own data property named P of object O whose [[Value]], [[Writable]],

// [[Enumerable]] and [[Configurable]] attribute values are described by Desc.

// If the value of an attribute field of Desc is absent, the attribute of the newly created property

// is set to its default value.

PropertyDescriptor filledDescriptor = FillMissingPropertyDescriptorFields<false>(descriptor, scriptContext);

BOOL tempResult = obj->SetPropertyWithAttributes(propId, filledDescriptor.GetValue(), filledDescriptor.GetAttributes(), nullptr);

Assert(tempResult || obj->IsExternal());

}

else

{

// ES5 8.12.9.4b: Create an own accessor property named P of object O whose [[Get]], [[Set]], [[Enumerable]]

// and [[Configurable]] attribute values are described by Desc. If the value of an attribute field of Desc is absent,

// the attribute of the newly created property is set to its default value.

Assert(descriptor.IsAccessorDescriptor());

PropertyDescriptor filledDescriptor = FillMissingPropertyDescriptorFields<true>(descriptor, scriptContext);

BOOL isSetAccessorsSuccess = obj->SetAccessors(propId, filledDescriptor.GetGetter(), filledDescriptor.GetSetter());

// It is valid for some objects to not-support getters and setters, specifically, for projection of an ABI method

// (CustomExternalObject => MapWithStringKey) which SetAccessors returns VBSErr\_ActionNotSupported.

// But for non-external objects SetAccessors should succeed.

Assert(isSetAccessorsSuccess || obj->CanHaveInterceptors());

// If SetAccessors failed, the property wasn't created, so no need to change the attributes.

if (isSetAccessorsSuccess)

{

JavascriptOperators::SetAttributes(obj, propId, filledDescriptor, true); // use 'force' as default attributes in type system are different from ES5.

}

}

}

return TRUE;

}

}

// ES5 8.12.9.5: Return true, if every field in Desc is absent.

if (!descriptor.ConfigurableSpecified() && !descriptor.EnumerableSpecified() && !descriptor.WritableSpecified() &&

!descriptor.ValueSpecified() && !descriptor.GetterSpecified() && !descriptor.SetterSpecified())

{

return TRUE;

}

// ES5 8.12.9.6: Return true, if every field in Desc also occurs in current and the value of every field in Desc is the same value

// as the corresponding field in current when compared using the SameValue algorithm (9.12).

PropertyDescriptor filledDescriptor = descriptor.IsAccessorDescriptor() ? FillMissingPropertyDescriptorFields<true>(descriptor, scriptContext)

: FillMissingPropertyDescriptorFields<false>(descriptor, scriptContext);

if (JavascriptOperators::AreSamePropertyDescriptors(&filledDescriptor, currentDescriptor, scriptContext))

{

return TRUE;

}

if (!currentDescriptor->IsConfigurable()) // ES5 8.12.9.7.

{

if (descriptor.ConfigurableSpecified() && descriptor.IsConfigurable())

{

return Reject(throwOnError, scriptContext, JSERR\_DefineProperty\_NotConfigurable, propId);

}

if (descriptor.EnumerableSpecified() && descriptor.IsEnumerable() != currentDescriptor->IsEnumerable())

{

return Reject(throwOnError, scriptContext, JSERR\_DefineProperty\_NotConfigurable, propId);

}

}

// Whether to merge attributes from tempDescriptor into descriptor to keep original values

// of some attributes from the object/use tempDescriptor for SetAttributes, or just use descriptor.

// This is optimization to avoid 2 calls to SetAttributes.

bool mergeDescriptors = false;

// Whether to call SetAttributes with 'force' flag which forces setting all attributes

// rather than only specified or which have true values.

// This is to make sure that the object has correct attributes, as default values in the object are not for ES5.

bool forceSetAttributes = false;

PropertyDescriptor tempDescriptor;

// ES5 8.12.9.8: If IsGenericDescriptor(Desc) is true, then no further validation is required.

if (!descriptor.IsGenericDescriptor())

{

if (currentDescriptor->IsDataDescriptor() != descriptor.IsDataDescriptor())

{

// ES5 8.12.9.9: Else, if IsDataDescriptor(current) and IsDataDescriptor(Desc) have different results...

if (!currentDescriptor->IsConfigurable())

{

return Reject(throwOnError, scriptContext, JSERR\_DefineProperty\_NotConfigurable, propId);

}

if (needToSetProperty)

{

if (currentDescriptor->IsDataDescriptor())

{

// ES5 8.12.9.9.b: Convert the property named P of object O from a data property to an accessor property.

// Preserve the existing values of the converted property's [[Configurable]] and [[Enumerable]] attributes

// and set the rest of the property's attributes to their default values.

PropertyAttributes preserveFromObject = currentDescriptor->GetAttributes() & (PropertyConfigurable | PropertyEnumerable);

BOOL isSetAccessorsSuccess = obj->SetAccessors(propId, defaultAccessorValue, defaultAccessorValue);

// It is valid for some objects to not-support getters and setters, specifically, for projection of an ABI method

// (CustomExternalObject => MapWithStringKey) which SetAccessors returns VBSErr\_ActionNotSupported.

// But for non-external objects SetAccessors should succeed.

Assert(isSetAccessorsSuccess || obj->CanHaveInterceptors());

if (isSetAccessorsSuccess)

{

tempDescriptor.SetAttributes(preserveFromObject, PropertyConfigurable | PropertyEnumerable);

forceSetAttributes = true; // use SetAttrbiutes with 'force' as default attributes in type system are different from ES5.

mergeDescriptors = true;

}

}

else

{

// ES5 8.12.9.9.c: Convert the property named P of object O from an accessor property to a data property.

// Preserve the existing values of the converted property's [[Configurable]] and [[Enumerable]] attributes

// and set the rest of the property's attributes to their default values.

// Note: avoid using SetProperty/SetPropertyWithAttributes here because they has undesired side-effects:

// it calls previous setter and in some cases of attribute values throws.

// To walk around, call DeleteProperty and then AddProperty.

PropertyAttributes preserveFromObject = currentDescriptor->GetAttributes() & (PropertyConfigurable | PropertyEnumerable);

tempDescriptor.SetAttributes(preserveFromObject, PropertyConfigurable | PropertyEnumerable);

tempDescriptor.MergeFrom(descriptor); // Update only fields specified in 'descriptor'.

Var descriptorValue = descriptor.ValueSpecified() ? descriptor.GetValue() : defaultDataValue;

// Note: HostDispath'es implementation of DeleteProperty currently throws E\_NOTIMPL.

obj->DeleteProperty(propId, PropertyOperation\_None);

BOOL tempResult = obj->SetPropertyWithAttributes(propId, descriptorValue, tempDescriptor.GetAttributes(), NULL, PropertyOperation\_Force);

Assert(tempResult);

// At this time we already set value and attributes to desired values,

// thus we can skip step ES5 8.12.9.12 and simply return true.

return TRUE;

}

}

}

else if (currentDescriptor->IsDataDescriptor() && descriptor.IsDataDescriptor())

{

// ES5 8.12.9.10: Else, if IsDataDescriptor(current) and IsDataDescriptor(Desc) are both true...

if (!currentDescriptor->IsConfigurable())

{

if (!currentDescriptor->IsWritable())

{

if ((descriptor.WritableSpecified() && descriptor.IsWritable()) || // ES5 8.12.9.10.a.i

(descriptor.ValueSpecified() &&

!JavascriptConversion::SameValue(descriptor.GetValue(), currentDescriptor->GetValue()))) // ES5 8.12.9.10.a.ii

{

return Reject(throwOnError, scriptContext, JSERR\_DefineProperty\_NotWritable, propId);

}

}

}

// ES5 8.12.9.10.b: else, the [[Configurable]] field of current is true, so any change is acceptable.

}

else

{

// ES5 8.12.9.11: Else, IsAccessorDescriptor(current) and IsAccessorDescriptor(Desc) are both true, so...

Assert(currentDescriptor->IsAccessorDescriptor() && descriptor.IsAccessorDescriptor());

if (!currentDescriptor->IsConfigurable())

{

if ((descriptor.SetterSpecified() &&

!JavascriptConversion::SameValue(

JavascriptOperators::CanonicalizeAccessor(descriptor.GetSetter(), scriptContext),

JavascriptOperators::CanonicalizeAccessor(currentDescriptor->GetSetter(), scriptContext))) ||

(descriptor.GetterSpecified() &&

!JavascriptConversion::SameValue(

JavascriptOperators::CanonicalizeAccessor(descriptor.GetGetter(), scriptContext),

JavascriptOperators::CanonicalizeAccessor(currentDescriptor->GetGetter(), scriptContext))))

{

return Reject(throwOnError, scriptContext, JSERR\_DefineProperty\_NotConfigurable, propId);

}

}

}

// This part is only for non-generic descriptors:

// ES5 8.12.9.12: For each attribute field of Desc that is present,

// set the correspondingly named attribute of the property named P of object O to the value of the field.

if (descriptor.IsDataDescriptor())

{

if (descriptor.ValueSpecified() && needToSetProperty)

{

// Set just the value by passing the current attributes of the property.

// If the property's attributes are also changing (perhaps becoming non-writable),

// this will be taken care of in the call to JavascriptOperators::SetAttributes below.

// Built-in Function.prototype properties 'length', 'arguments', and 'caller' are special cases.

BOOL tempResult = obj->SetPropertyWithAttributes(propId, descriptor.GetValue(), currentDescriptor->GetAttributes(), nullptr);

AssertMsg(tempResult || JavascriptFunction::IsBuiltinProperty(obj, propId), "If you hit this assert, most likely there is something wrong with the object/type.");

}

}

else if (descriptor.IsAccessorDescriptor() && needToSetProperty)

{

Assert(descriptor.GetterSpecified() || descriptor.SetterSpecified());

Var oldGetter = defaultAccessorValue, oldSetter = defaultAccessorValue;

if (!descriptor.GetterSpecified() || !descriptor.SetterSpecified())

{

// Unless both getter and setter are specified, make sure we don't overwrite old accessor.

obj->GetAccessors(propId, &oldGetter, &oldSetter, scriptContext);

}

Var getter = descriptor.GetterSpecified() ? descriptor.GetGetter() : oldGetter;

Var setter = descriptor.SetterSpecified() ? descriptor.GetSetter() : oldSetter;

obj->SetAccessors(propId, getter, setter);

}

} // if (!descriptor.IsGenericDescriptor())

// Continue for all descriptors including generic:

// ES5 8.12.9.12: For each attribute field of Desc that is present,

// set the correspondingly named attribute of the property named P of object O to the value of the field.

if (needToSetProperty)

{

if (mergeDescriptors)

{

tempDescriptor.MergeFrom(descriptor);

JavascriptOperators::SetAttributes(obj, propId, tempDescriptor, forceSetAttributes);

}

else

{

JavascriptOperators::SetAttributes(obj, propId, descriptor, forceSetAttributes);

}

}

return TRUE;

}

template <bool isAccessor>

PropertyDescriptor JavascriptOperators::FillMissingPropertyDescriptorFields(PropertyDescriptor descriptor, ScriptContext\* scriptContext)

{

PropertyDescriptor newDescriptor;

const PropertyDescriptor\* defaultDescriptor = scriptContext->GetLibrary()->GetDefaultPropertyDescriptor();

if (isAccessor)

{

newDescriptor.SetGetter(descriptor.GetterSpecified() ? descriptor.GetGetter() : defaultDescriptor->GetGetter());

newDescriptor.SetSetter(descriptor.SetterSpecified() ? descriptor.GetSetter() : defaultDescriptor->GetSetter());

}

else

{

newDescriptor.SetValue(descriptor.ValueSpecified() ? descriptor.GetValue() : defaultDescriptor->GetValue());

newDescriptor.SetWritable(descriptor.WritableSpecified() ? descriptor.IsWritable() : defaultDescriptor->IsWritable());

}

newDescriptor.SetConfigurable(descriptor.ConfigurableSpecified() ? descriptor.IsConfigurable() : defaultDescriptor->IsConfigurable());

newDescriptor.SetEnumerable(descriptor.EnumerableSpecified() ? descriptor.IsEnumerable() : defaultDescriptor->IsEnumerable());

return newDescriptor;

}

// ES5: 15.4.5.1

BOOL JavascriptOperators::DefineOwnPropertyForArray(JavascriptArray\* arr, PropertyId propId, const PropertyDescriptor& descriptor, bool throwOnError, ScriptContext\* scriptContext)

{

if (propId == PropertyIds::length)

{

if (!descriptor.ValueSpecified())

{

return DefineOwnPropertyDescriptor(arr, PropertyIds::length, descriptor, throwOnError, scriptContext);

}

PropertyDescriptor newLenDesc = descriptor;

uint32 newLen = ES5Array::ToLengthValue(descriptor.GetValue(), scriptContext);

newLenDesc.SetValue(JavascriptNumber::ToVar(newLen, scriptContext));

uint32 oldLen = arr->GetLength();

if (newLen >= oldLen)

{

return DefineOwnPropertyDescriptor(arr, PropertyIds::length, newLenDesc, throwOnError, scriptContext);

}

BOOL oldLenWritable = arr->IsWritable(PropertyIds::length);

if (!oldLenWritable)

{

return Reject(throwOnError, scriptContext, JSERR\_DefineProperty\_NotWritable, propId);

}

bool newWritable = (!newLenDesc.WritableSpecified() || newLenDesc.IsWritable());

if (!newWritable)

{

// Need to defer setting writable to false in case any elements cannot be deleted

newLenDesc.SetWritable(true);

}

BOOL succeeded = DefineOwnPropertyDescriptor(arr, PropertyIds::length, newLenDesc, throwOnError, scriptContext);

//

// Our SetProperty(length) is also responsible to trim elements. When succeeded is

//

// false:

// \* length attributes rejected

// \* elements not touched

// true:

// \* length attributes are set successfully

// \* elements trimming may be either completed or incompleted, length value is correct

//

// \* Strict mode TODO: Currently SetProperty(length) does not throw. If that throws, we need

// to update here to set correct newWritable even on exception.

//

if (!succeeded)

{

return false;

}

if (!newWritable) // Now set requested newWritable.

{

PropertyDescriptor newWritableDesc;

newWritableDesc.SetWritable(false);

DefineOwnPropertyDescriptor(arr, PropertyIds::length, newWritableDesc, false, scriptContext);

}

if (arr->GetLength() > newLen) // Delete incompleted

{

// Since SetProperty(length) not throwing, we'll reject here

return Reject(throwOnError, scriptContext, JSERR\_DefineProperty\_Default, propId);

}

return true;

}

uint32 index;

if (scriptContext->IsNumericPropertyId(propId, &index))

{

if (index >= arr->GetLength() && !arr->IsWritable(PropertyIds::length))

{

return Reject(throwOnError, scriptContext, JSERR\_DefineProperty\_LengthNotWritable, propId);

}

BOOL succeeded = DefineOwnPropertyDescriptor(arr, propId, descriptor, false, scriptContext);

if (!succeeded)

{

return Reject(throwOnError, scriptContext, JSERR\_DefineProperty\_Default, propId);

}

// Out SetItem takes care of growing "length". we are done.

return true;

}

return DefineOwnPropertyDescriptor(arr, propId, descriptor, throwOnError, scriptContext);

}

BOOL JavascriptOperators::SetPropertyDescriptor(RecyclableObject\* object, PropertyId propId, PropertyDescriptor descriptor)

{

if (descriptor.ValueSpecified())

{

ScriptContext\* requestContext = object->GetScriptContext(); // Real requestContext?

JavascriptOperators::SetProperty(object, object, propId, descriptor.GetValue(), requestContext);

}

else if (descriptor.GetterSpecified() || descriptor.SetterSpecified())

{

JavascriptOperators::SetAccessors(object, propId, descriptor.GetGetter(), descriptor.GetSetter());

}

if (descriptor.EnumerableSpecified())

{

object->SetEnumerable(propId, descriptor.IsEnumerable());

}

if (descriptor.ConfigurableSpecified())

{

object->SetConfigurable(propId, descriptor.IsConfigurable());

}

if (descriptor.WritableSpecified())

{

object->SetWritable(propId, descriptor.IsWritable());

}

return true;

}

BOOL JavascriptOperators::ToPropertyDescriptorForProxyObjects(Var propertySpec, PropertyDescriptor\* descriptor, ScriptContext\* scriptContext)

{

if (!JavascriptOperators::IsObject(propertySpec))

{

return FALSE;

}

Var value;

RecyclableObject\* propertySpecObj = RecyclableObject::FromVar(propertySpec);

if (JavascriptOperators::HasProperty(propertySpecObj, PropertyIds::enumerable) == TRUE)

{

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::enumerable, &value, scriptContext))

{

descriptor->SetEnumerable(JavascriptConversion::ToBoolean(value, scriptContext) ? true : false);

}

else

{

AssertMsg(FALSE, "Proxy : HasProperty and GetProperty's result don't match for 'enumerable'.");

}

}

if (JavascriptOperators::HasProperty(propertySpecObj, PropertyIds::configurable) == TRUE)

{

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::configurable, &value, scriptContext))

{

descriptor->SetConfigurable(JavascriptConversion::ToBoolean(value, scriptContext) ? true : false);

}

else

{

AssertMsg(FALSE, "Proxy : HasProperty and GetProperty's result don't match for 'configurable'.");

}

}

if (JavascriptOperators::HasProperty(propertySpecObj, PropertyIds::value) == TRUE)

{

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::value, &value, scriptContext))

{

descriptor->SetValue(value);

}

else

{

AssertMsg(FALSE, "Proxy : HasProperty and GetProperty's result don't match for 'value'.");

}

}

if (JavascriptOperators::HasProperty(propertySpecObj, PropertyIds::writable) == TRUE)

{

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::writable, &value, scriptContext))

{

descriptor->SetWritable(JavascriptConversion::ToBoolean(value, scriptContext) ? true : false);

}

else

{

AssertMsg(FALSE, "Proxy : HasProperty and GetProperty's result don't match for 'writable'.");

}

}

if (JavascriptOperators::HasProperty(propertySpecObj, PropertyIds::get) == TRUE)

{

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::get, &value, scriptContext))

{

if (JavascriptOperators::GetTypeId(value) != TypeIds\_Undefined && (false == JavascriptConversion::IsCallable(value)))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_NeedFunction, scriptContext->GetPropertyName(PropertyIds::get)->GetBuffer());

}

descriptor->SetGetter(value);

}

else

{

AssertMsg(FALSE, "Proxy : HasProperty and GetProperty's result don't match for 'get'.");

}

}

if (JavascriptOperators::HasProperty(propertySpecObj, PropertyIds::set) == TRUE)

{

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::set, &value, scriptContext))

{

if (JavascriptOperators::GetTypeId(value) != TypeIds\_Undefined && (false == JavascriptConversion::IsCallable(value)))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_NeedFunction, scriptContext->GetPropertyName(PropertyIds::set)->GetBuffer());

}

descriptor->SetSetter(value);

}

else

{

AssertMsg(FALSE, "Proxy : HasProperty and GetProperty's result don't match for 'set'.");

}

}

return TRUE;

}

BOOL JavascriptOperators::ToPropertyDescriptorForGenericObjects(Var propertySpec, PropertyDescriptor\* descriptor, ScriptContext\* scriptContext)

{

if (!JavascriptOperators::IsObject(propertySpec))

{

return FALSE;

}

Var value;

RecyclableObject\* propertySpecObj = RecyclableObject::FromVar(propertySpec);

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::enumerable, &value, scriptContext))

{

descriptor->SetEnumerable(JavascriptConversion::ToBoolean(value, scriptContext) ? true : false);

}

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::configurable, &value, scriptContext))

{

descriptor->SetConfigurable(JavascriptConversion::ToBoolean(value, scriptContext) ? true : false);

}

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::value, &value, scriptContext))

{

descriptor->SetValue(value);

}

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::writable, &value, scriptContext))

{

descriptor->SetWritable(JavascriptConversion::ToBoolean(value, scriptContext) ? true : false);

}

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::get, &value, scriptContext))

{

if (JavascriptOperators::GetTypeId(value) != TypeIds\_Undefined && (false == JavascriptConversion::IsCallable(value)))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_NeedFunction, scriptContext->GetPropertyName(PropertyIds::get)->GetBuffer());

}

descriptor->SetGetter(value);

}

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::set, &value, scriptContext))

{

if (JavascriptOperators::GetTypeId(value) != TypeIds\_Undefined && (false == JavascriptConversion::IsCallable(value)))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_NeedFunction, scriptContext->GetPropertyName(PropertyIds::set)->GetBuffer());

}

descriptor->SetSetter(value);

}

return TRUE;

}

BOOL JavascriptOperators::ToPropertyDescriptor(Var propertySpec, PropertyDescriptor\* descriptor, ScriptContext\* scriptContext)

{

if (JavascriptProxy::Is(propertySpec))

{

if (ToPropertyDescriptorForProxyObjects(propertySpec, descriptor, scriptContext) == FALSE)

{

return FALSE;

}

}

else

{

if (ToPropertyDescriptorForGenericObjects(propertySpec, descriptor, scriptContext) == FALSE)

{

return FALSE;

}

}

if (descriptor->GetterSpecified() || descriptor->SetterSpecified())

{

if (descriptor->ValueSpecified())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotHaveAccessorsAndValue);

}

if (descriptor->WritableSpecified())

{

long hCode = descriptor->IsWritable() ? JSERR\_InvalidAttributeTrue : JSERR\_InvalidAttributeFalse;

JavascriptError::ThrowTypeError(scriptContext, hCode, L"writable");

}

}

descriptor->SetOriginal(propertySpec);

return TRUE;

}

void JavascriptOperators::CompletePropertyDescriptor(PropertyDescriptor\* resultDescriptor, PropertyDescriptor\* likeDescriptor, ScriptContext\* requestContext)

{

const PropertyDescriptor\* likePropertyDescriptor = likeDescriptor;

// 1. Assert: LikeDesc is either a Property Descriptor or undefined.

// 2. ReturnIfAbrupt(Desc).

// 3. Assert : Desc is a Property Descriptor

// 4. If LikeDesc is undefined, then set LikeDesc to Record{ [[Value]]: undefined, [[Writable]] : false, [[Get]] : undefined, [[Set]] : undefined, [[Enumerable]] : false, [[Configurable]] : false }.

if (likePropertyDescriptor == nullptr)

{

likePropertyDescriptor = requestContext->GetLibrary()->GetDefaultPropertyDescriptor();

}

// 5. If either IsGenericDescriptor(Desc) or IsDataDescriptor(Desc) is true, then

if (resultDescriptor->IsDataDescriptor() || resultDescriptor->IsGenericDescriptor())

{

// a.If Desc does not have a[[Value]] field, then set Desc.[[Value]] to LikeDesc.[[Value]].

// b.If Desc does not have a[[Writable]] field, then set Desc.[[Writable]] to LikeDesc.[[Writable]].

if (!resultDescriptor->ValueSpecified())

{

resultDescriptor->SetValue(likePropertyDescriptor->GetValue());

}

if (!resultDescriptor->WritableSpecified())

{

resultDescriptor->SetWritable(likePropertyDescriptor->IsWritable());

}

}

else

{

// 6. Else,

// a.If Desc does not have a[[Get]] field, then set Desc.[[Get]] to LikeDesc.[[Get]].

// b.If Desc does not have a[[Set]] field, then set Desc.[[Set]] to LikeDesc.[[Set]].

if (!resultDescriptor->GetterSpecified())

{

resultDescriptor->SetGetter(likePropertyDescriptor->GetGetter());

}

if (!resultDescriptor->SetterSpecified())

{

resultDescriptor->SetSetter(likePropertyDescriptor->GetSetter());

}

}

// 7. If Desc does not have an[[Enumerable]] field, then set Desc.[[Enumerable]] to LikeDesc.[[Enumerable]].

// 8. If Desc does not have a[[Configurable]] field, then set Desc.[[Configurable]] to LikeDesc.[[Configurable]].

// 9. Return Desc.

if (!resultDescriptor->EnumerableSpecified())

{

resultDescriptor->SetEnumerable(likePropertyDescriptor->IsEnumerable());

}

if (!resultDescriptor->ConfigurableSpecified())

{

resultDescriptor->SetConfigurable(likePropertyDescriptor->IsConfigurable());

}

}

Var JavascriptOperators::OP\_InvokePut(Js::ScriptContext \*scriptContext, Var instance, CallInfo callInfo, ...)

{

// Handle a store to a call result: x(y) = z.

// This is not strictly permitted in JScript, but some scripts expect to be able to use

// the syntax to set properties of ActiveX objects.

// We handle this by deferring to a virtual method of type. This incurs an extra level of

// indirection but seems preferable to adding the "put" method as a member of every type

// and using the normal JScript calling mechanism.

RUNTIME\_ARGUMENTS(args, callInfo);

AssertMsg(args.Info.Count > 0, "Missing this argument in InvokePut");

if (TaggedNumber::Is(instance))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedFunction /\* TODO-ERROR: get arg name - aFunc \*/);

}

RecyclableObject\* function = RecyclableObject::FromVar(instance);

return function->InvokePut(args);

}

// Conformance to: ES5 8.6.1.

// Set attributes on the object as provided by property descriptor.

// If force parameter is true, we force SetAttributes call even if none of the attributes are defined by the descriptor.

// NOTE: does not set [[Get]], [Set]], [[Value]]

void JavascriptOperators::SetAttributes(RecyclableObject\* object, PropertyId propId, const PropertyDescriptor& descriptor, bool force)

{

Assert(object);

BOOL isWritable = FALSE;

if (descriptor.IsDataDescriptor())

{

isWritable = descriptor.WritableSpecified() ? descriptor.IsWritable() : FALSE;

}

else if (descriptor.IsAccessorDescriptor())

{

// The reason is that JavascriptOperators::OP\_SetProperty checks for RecyclableObject::FromVar(instance)->IsWritableOrAccessor(propertyId),

// which should in fact check for 'is writable or accessor' but since there is no GetAttributes, we can't do that efficiently.

isWritable = TRUE;

}

// CONSIDER: call object->SetAttributes which is much more efficient as that's 1 call instead of 3.

// Can't do that now as object->SetAttributes doesn't provide a way which attributes to modify and which not.

if (force || descriptor.ConfigurableSpecified())

{

object->SetConfigurable(propId, descriptor.ConfigurableSpecified() ? descriptor.IsConfigurable() : FALSE);

}

if (force || descriptor.EnumerableSpecified())

{

object->SetEnumerable(propId, descriptor.EnumerableSpecified() ? descriptor.IsEnumerable() : FALSE);

}

if (force || descriptor.WritableSpecified() || isWritable)

{

object->SetWritable(propId, isWritable);

}

}

void JavascriptOperators::OP\_ClearAttributes(Var instance, PropertyId propertyId)

{

Assert(instance);

if (RecyclableObject::Is(instance))

{

RecyclableObject\* obj = RecyclableObject::FromVar(instance);

obj->SetAttributes(propertyId, PropertyNone);

}

}

void JavascriptOperators::OP\_Freeze(Var instance)

{

Assert(instance);

if (RecyclableObject::Is(instance))

{

RecyclableObject\* obj = RecyclableObject::FromVar(instance);

obj->Freeze();

}

}

BOOL JavascriptOperators::Reject(bool throwOnError, ScriptContext\* scriptContext, long errorCode, PropertyId propertyId)

{

Assert(scriptContext);

if (throwOnError)

{

JavascriptError::ThrowTypeError(scriptContext, errorCode, scriptContext->GetThreadContext()->GetPropertyName(propertyId)->GetBuffer());

}

return FALSE;

}

bool JavascriptOperators::AreSamePropertyDescriptors(const PropertyDescriptor\* x, const PropertyDescriptor\* y, ScriptContext\* scriptContext)

{

Assert(scriptContext);

if (x->ConfigurableSpecified() != y->ConfigurableSpecified() || x->IsConfigurable() != y->IsConfigurable() ||

x->EnumerableSpecified() != y->EnumerableSpecified() || x->IsEnumerable() != y->IsEnumerable())

{

return false;

}

if (x->IsDataDescriptor())

{

if (!y->IsDataDescriptor() || x->WritableSpecified() != y->WritableSpecified() || x->IsWritable() != y->IsWritable())

{

return false;

}

if (x->ValueSpecified())

{

if (!y->ValueSpecified() || !JavascriptConversion::SameValue(x->GetValue(), y->GetValue()))

{

return false;

}

}

}

else if (x->IsAccessorDescriptor())

{

if (!y->IsAccessorDescriptor())

{

return false;

}

if (x->GetterSpecified())

{

if (!y->GetterSpecified() || !JavascriptConversion::SameValue(

JavascriptOperators::CanonicalizeAccessor(x->GetGetter(), scriptContext),

JavascriptOperators::CanonicalizeAccessor(y->GetGetter(), scriptContext)))

{

return false;

}

}

if (x->SetterSpecified())

{

if (!y->SetterSpecified() || !JavascriptConversion::SameValue(

JavascriptOperators::CanonicalizeAccessor(x->GetSetter(), scriptContext),

JavascriptOperators::CanonicalizeAccessor(y->GetSetter(), scriptContext)))

{

return false;

}

}

}

return true;

}

// Check if an accessor is undefined (null or defaultAccessor)

bool JavascriptOperators::IsUndefinedAccessor(Var accessor, ScriptContext\* scriptContext)

{

return nullptr == accessor || scriptContext->GetLibrary()->GetDefaultAccessorFunction() == accessor;

}

// Converts default accessor to undefined.

// Can be used when comparing accessors.

Var JavascriptOperators::CanonicalizeAccessor(Var accessor, ScriptContext\* scriptContext)

{

Assert(scriptContext);

if (IsUndefinedAccessor(accessor, scriptContext))

{

return scriptContext->GetLibrary()->GetUndefined();

}

return accessor;

}

Var JavascriptOperators::DefaultAccessor(RecyclableObject\* function, CallInfo callInfo, ...)

{

return function->GetLibrary()->GetUndefined();

}

void FrameDisplay::SetItem(uint index, void\* item)

{

AssertMsg(index < this->length, "Invalid frame display access");

scopes[index] = item;

}

void \*FrameDisplay::GetItem(uint index)

{

AssertMsg(index < this->length, "Invalid frame display access");

return scopes[index];

}

// Grab the "this" pointer, mapping a root object to its associated host object.

Var JavascriptOperators::RootToThisObject(const Var object, ScriptContext\* scriptContext)

{

Js::Var thisVar = object;

TypeId typeId = Js::JavascriptOperators::GetTypeId(thisVar);

switch (typeId)

{

case Js::TypeIds\_GlobalObject:

return ((Js::GlobalObject\*)thisVar)->ToThis();

case Js::TypeIds\_ModuleRoot:

return Js::JavascriptOperators::GetThisFromModuleRoot(thisVar);

default:

if (typeId == scriptContext->GetDirectHostTypeId())

{

return ((RecyclableObject\*)thisVar)->GetLibrary()->GetGlobalObject()->ToThis();

}

}

return thisVar;

}

Var JavascriptOperators::CallGetter(RecyclableObject \* const function, Var const object, ScriptContext \* requestContext)

{

ScriptContext \* scriptContext = function->GetScriptContext();

ThreadContext \* threadContext = scriptContext->GetThreadContext();

return threadContext->ExecuteImplicitCall(function, ImplicitCall\_Accessor, [=]() -> Js::Var

{

// Stack object should have a pre-op bail on implicit call. We shouldn't see them here.

// Stack numbers are ok, as we will call ToObject to wrap it in a number object anyway

// See JavascriptOperators::GetThisHelper

Assert(JavascriptOperators::GetTypeId(object) == TypeIds\_Integer ||

JavascriptOperators::GetTypeId(object) == TypeIds\_Number || !ThreadContext::IsOnStack(object));

// Verify that the scriptcontext is alive before firing getter/setter

if (!scriptContext->VerifyAlive(!function->IsExternal(), requestContext))

{

return nullptr;

}

CallFlags flags = CallFlags\_Value;

Var thisVar = RootToThisObject(object, scriptContext);

RecyclableObject\* marshalledFunction = RecyclableObject::FromVar(CrossSite::MarshalVar(requestContext, function));

Var result = marshalledFunction->GetEntryPoint()(function, CallInfo(flags, 1), thisVar);

result = CrossSite::MarshalVar(requestContext, result);

return result;

});

}

void JavascriptOperators::CallSetter(RecyclableObject \* const function, Var const object, Var const value, ScriptContext \* requestContext)

{

ScriptContext \* scriptContext = function->GetScriptContext();

ThreadContext \* threadContext = scriptContext->GetThreadContext();

threadContext->ExecuteImplicitCall(function, ImplicitCall\_Accessor, [=]() -> Js::Var

{

// Stack object should have a pre-op bail on implicit call. We shouldn't see them here.

// Stack numbers are ok, as we will call ToObject to wrap it in a number object anyway

// See JavascriptOperators::GetThisHelper

Assert(JavascriptOperators::GetTypeId(object) == TypeIds\_Integer ||

JavascriptOperators::GetTypeId(object) == TypeIds\_Number || !ThreadContext::IsOnStack(object));

// Verify that the scriptcontext is alive before firing getter/setter

if (!scriptContext->VerifyAlive(!function->IsExternal(), requestContext))

{

return nullptr;

}

CallFlags flags = CallFlags\_Value;

Var putValue = value;

// CONSIDER: Have requestContext everywhere, even in the setProperty related codepath.

if (requestContext)

{

putValue = CrossSite::MarshalVar(requestContext, value);

}

Var thisVar = RootToThisObject(object, scriptContext);

RecyclableObject\* marshalledFunction = function;

if (requestContext)

{

marshalledFunction = RecyclableObject::FromVar(CrossSite::MarshalVar(requestContext, function));

}

Var result = marshalledFunction->GetEntryPoint()(function, CallInfo(flags, 2), thisVar, putValue);

Assert(result);

return nullptr;

});

}

void \* JavascriptOperators::AllocMemForVarArray(size\_t size, Recycler\* recycler)

{

TRACK\_ALLOC\_INFO(recycler, Js::Var, Recycler, 0, (size\_t)(size / sizeof(Js::Var)));

return recycler->AllocZero(size);

}

void \* JavascriptOperators::AllocUninitializedNumber(Js::RecyclerJavascriptNumberAllocator \* allocator)

{

TRACK\_ALLOC\_INFO(allocator->GetRecycler(), Js::JavascriptNumber, Recycler, 0, (size\_t)-1);

return allocator->Alloc(sizeof(Js::JavascriptNumber));

}

void JavascriptOperators::ScriptAbort()

{

throw ScriptAbortException();

}

void PolymorphicInlineCache::Finalize(bool isShutdown)

{

if (size == 0)

{

// Already finalized

Assert(!inlineCaches && !prev && !next);

return;

}

Assert(inlineCaches && size > 0);

// If we're not shutting down (as in closing the script context), we need to remove our inline caches from

// thread context's invalidation lists, and release memory back to the arena. During script context shutdown,

// we leave everything in place, because the inline cache arena will stay alive until script context is destroyed

// (as in destructor has been called) and thus the invalidation lists are safe to keep references to caches from this

// script context. We will, however, zero all inline caches so that we don't have to process them on subsequent

// collections, which may still happen from other script contexts.

if (isShutdown)

{

memset(inlineCaches, 0, size \* sizeof(InlineCache));

}

else

{

for (int i = 0; i < size; i++)

{

inlineCaches[i].RemoveFromInvalidationList();

}

AllocatorDeleteArray(InlineCacheAllocator, functionBody->GetScriptContext()->GetInlineCacheAllocator(), size, inlineCaches);

#ifdef POLY\_INLINE\_CACHE\_SIZE\_STATS

functionBody->GetScriptContext()->GetInlineCacheAllocator()->LogPolyCacheFree(size \* sizeof(InlineCache));

#endif

}

// Remove this PolymorphicInlineCache from the list

if (this == functionBody->GetPolymorphicInlineCachesHead())

{

Assert(!prev);

if (next)

{

Assert(next->prev == this);

next->prev = nullptr;

}

functionBody->SetPolymorphicInlineCachesHead(next);

}

else

{

if (prev)

{

Assert(prev->next == this);

prev->next = next;

}

if (next)

{

Assert(next->prev == this);

next->prev = prev;

}

}

prev = next = nullptr;

inlineCaches = nullptr;

size = 0;

}

JavascriptString \* JavascriptOperators::Concat3(Var aLeft, Var aCenter, Var aRight, ScriptContext \* scriptContext)

{

// Make sure we do the conversion in order from left to right

JavascriptString \* strLeft = JavascriptConversion::ToPrimitiveString(aLeft, scriptContext);

JavascriptString \* strCenter = JavascriptConversion::ToPrimitiveString(aCenter, scriptContext);

JavascriptString \* strRight = JavascriptConversion::ToPrimitiveString(aRight, scriptContext);

return JavascriptString::Concat3(strLeft, strCenter, strRight);

}

JavascriptString \*

JavascriptOperators::NewConcatStrMulti(Var a1, Var a2, uint count, ScriptContext \* scriptContext)

{

// Make sure we do the conversion in order

JavascriptString \* str1 = JavascriptConversion::ToPrimitiveString(a1, scriptContext);

JavascriptString \* str2 = JavascriptConversion::ToPrimitiveString(a2, scriptContext);

return ConcatStringMulti::New(count, str1, str2, scriptContext);

}

void

JavascriptOperators::SetConcatStrMultiItem(Var concatStr, Var str, uint index, ScriptContext \* scriptContext)

{

ConcatStringMulti::FromVar(concatStr)->SetItem(index,

JavascriptConversion::ToPrimitiveString(str, scriptContext));

}

void

JavascriptOperators::SetConcatStrMultiItem2(Var concatStr, Var str1, Var str2, uint index, ScriptContext \* scriptContext)

{

ConcatStringMulti \* cs = ConcatStringMulti::FromVar(concatStr);

cs->SetItem(index, JavascriptConversion::ToPrimitiveString(str1, scriptContext));

cs->SetItem(index + 1, JavascriptConversion::ToPrimitiveString(str2, scriptContext));

}

void JavascriptOperators::OP\_SetComputedNameVar(Var method, Var computedNameVar)

{

ScriptFunctionBase \*scriptFunction = ScriptFunctionBase::FromVar(method);

scriptFunction->SetComputedNameVar(computedNameVar);

}

void JavascriptOperators::OP\_SetHomeObj(Var method, Var homeObj)

{

ScriptFunctionBase \*scriptFunction = ScriptFunctionBase::FromVar(method);

scriptFunction->SetHomeObj(homeObj);

}

Var JavascriptOperators::OP\_LdSuper(Var scriptFunction, ScriptContext \* scriptContext)

{

// Ensure this is not a stack ScriptFunction

if (!ScriptFunction::Is(scriptFunction) || ThreadContext::IsOnStack(scriptFunction))

{

return scriptContext->GetLibrary()->GetUndefined();

}

ScriptFunction \*instance = ScriptFunction::FromVar(scriptFunction);

// We keep a reference to the current class rather than its super prototype

// since the prototype could change.

Var homeObj = instance->GetHomeObj();

if (homeObj == nullptr || !RecyclableObject::Is(homeObj))

{

return scriptContext->GetLibrary()->GetUndefined();

}

RecyclableObject \*thisObjPrototype = RecyclableObject::FromVar(homeObj);

Assert(thisObjPrototype != nullptr);

RecyclableObject \*superBase = thisObjPrototype->GetPrototype();

if (superBase == nullptr || !RecyclableObject::Is(superBase))

{

return scriptContext->GetLibrary()->GetUndefined();

}

return superBase;

}

Var JavascriptOperators::OP\_LdSuperCtor(Var scriptFunction, ScriptContext \* scriptContext)

{

// use self as value of [[FunctionObject]] - this is true only for constructors

Assert(RecyclableObject::Is(scriptFunction));

Assert(JavascriptOperators::IsClassConstructor(scriptFunction)); // non-constructors cannot have direct super

RecyclableObject \*superCtor = RecyclableObject::FromVar(scriptFunction)->GetPrototype();

if (superCtor == nullptr || !IsConstructor(superCtor))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NotAConstructor, L"super");

}

return superCtor;

}

Var JavascriptOperators::ScopedLdSuperHelper(Var scriptFunction, Js::PropertyId propertyId, ScriptContext \* scriptContext)

{

ScriptFunction \*instance = ScriptFunction::FromVar(scriptFunction);

Var superRef = nullptr;

FrameDisplay \*frameDisplay = instance->GetEnvironment();

if (frameDisplay->GetLength() == 0)

{

// Globally scoped evals are a syntax error

JavascriptError::ThrowSyntaxError(scriptContext, ERRSuperInGlobalEval, L"super");

}

// Iterate over the scopes in the FrameDisplay, looking for the super property.

for (unsigned i = 0; i < frameDisplay->GetLength(); ++i)

{

void \*currScope = frameDisplay->GetItem(i);

if (RecyclableObject::Is(currScope))

{

if (BlockActivationObject::Is(currScope))

{

// We won't find super in a block scope.

continue;

}

RecyclableObject \*recyclableObject = RecyclableObject::FromVar(currScope);

if (GetProperty(recyclableObject, propertyId, &superRef, scriptContext))

{

return superRef;

}

if (HasProperty(recyclableObject, Js::PropertyIds::\_lexicalThisSlotSymbol))

{

// If we reach 'this' and haven't found the super reference, we don't need to look any further.

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_BadSuperReference, L"super");

}

}

}

if (superRef == nullptr)

{

// We didn't find a super reference. Emit a reference error.

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_BadSuperReference, L"super");

}

return superRef;

}

Var JavascriptOperators::OP\_ScopedLdSuper(Var scriptFunction, ScriptContext \* scriptContext)

{

return JavascriptOperators::ScopedLdSuperHelper(scriptFunction, Js::PropertyIds::\_superReferenceSymbol, scriptContext);

}

Var JavascriptOperators::OP\_ScopedLdSuperCtor(Var scriptFunction, ScriptContext \* scriptContext)

{

return JavascriptOperators::ScopedLdSuperHelper(scriptFunction, Js::PropertyIds::\_superCtorReferenceSymbol, scriptContext);

}

Var JavascriptOperators::OP\_ResumeYield(ResumeYieldData\* yieldData, RecyclableObject\* iterator)

{

// CONSIDER: Fast path this early out return path in JITed code before helper call to avoid the helper call overhead in the common case e.g. next() calls.

if (yieldData->exceptionObj == nullptr)

{

return yieldData->data;

}

ScriptContext\* scriptContext = yieldData->exceptionObj->GetScriptContext();

bool isReturn = yieldData->exceptionObj->IsGeneratorReturnException();

if (iterator != nullptr)

{

PropertyId propertyId = isReturn ? PropertyIds::return\_ : PropertyIds::throw\_;

Var prop = nullptr;

Var args[] = { iterator, yieldData->data };

CallInfo callInfo(CallFlags\_Value, \_countof(args));

if (JavascriptOperators::GetProperty(iterator, iterator, propertyId, &prop, iterator->GetScriptContext())

&& prop != iterator->GetLibrary()->GetUndefined())

{

RecyclableObject\* method = RecyclableObject::FromVar(prop);

Var result = JavascriptFunction::CallFunction<true>(method, method->GetEntryPoint(), Arguments(callInfo, args));

if (isReturn)

{

if (!JavascriptOperators::IsObject(result))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedObject);

}

Var value = JavascriptOperators::GetProperty(RecyclableObject::FromVar(result), PropertyIds::value, scriptContext);

// CONSIDER: Using an exception to carry the return value and force finally code to execute is a bit of a janky

// solution since we have to override the value here in the case of yield\* expressions. It works but is there

// a more elegant way?

//

// Instead what if ResumeYield was a "set Dst then optionally branch" opcode, that could also throw? Then we could

// avoid using a special exception entirely with byte code something like this:

//

// ;; Ry is the yieldData

//

// ResumeYield Rx Ry $returnPathLabel

// ... code like normal

// $returnPathLabel:

// Ld\_A R0 Rx

// Br $exitFinallyAndReturn

//

// This would probably give better performance for the common case of calling next() on generators since we wouldn't

// have to wrap the call to the generator code in a try catch.

yieldData->exceptionObj->SetThrownObject(value);

}

}

else if (!isReturn)

{

// Throw is called on yield\* but the iterator does not have a throw method. This is a protocol violation.

// So we have to call IteratorClose().

if (JavascriptOperators::GetProperty(iterator, iterator, PropertyIds::return\_, &prop, iterator->GetScriptContext())

&& prop != iterator->GetLibrary()->GetUndefined())

{

// As per the spec we ignore the inner result after checking whether it is a valid object

RecyclableObject\* method = RecyclableObject::FromVar(prop);

Var result = JavascriptFunction::CallFunction<true>(method, method->GetEntryPoint(), Arguments(callInfo, args));

if (!JavascriptOperators::IsObject(result))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedObject);

}

}

}

}

if (!isReturn)

{

// Use ThrowExceptionObject() to get debugger support for breaking on throw

JavascriptExceptionOperators::ThrowExceptionObject(yieldData->exceptionObj, scriptContext, true);

}

// Do not use ThrowExceptionObject for return() API exceptions since these exceptions are not real exceptions

throw yieldData->exceptionObj;

}

Var JavascriptOperators::OP\_AsyncSpawn(Var aGenerator, Var aThis, ScriptContext\* scriptContext)

{

JavascriptLibrary\* library = scriptContext->GetLibrary();

JavascriptExceptionObject\* e = nullptr;

JavascriptPromiseResolveOrRejectFunction\* resolve;

JavascriptPromiseResolveOrRejectFunction\* reject;

JavascriptPromiseAsyncSpawnExecutorFunction\* executor = library->CreatePromiseAsyncSpawnExecutorFunction(JavascriptPromise::EntryJavascriptPromiseAsyncSpawnExecutorFunction, (JavascriptGenerator\*)aGenerator, aThis);

JavascriptPromise\* promise = library->CreatePromise();

JavascriptPromise::InitializePromise(promise, &resolve, &reject, scriptContext);

try

{

executor->GetEntryPoint()(executor, CallInfo(CallFlags\_Value, 3), library->GetUndefined(), resolve, reject);

}

catch (JavascriptExceptionObject\* ex)

{

e = ex;

}

if (e != nullptr)

{

reject->GetEntryPoint()(reject, CallInfo(CallFlags\_Value, 2), library->GetUndefined(), e->GetThrownObject(scriptContext));

}

return promise;

}

Js::Var

JavascriptOperators::BoxStackInstance(Js::Var instance, ScriptContext \* scriptContext, bool allowStackFunction)

{

if (!ThreadContext::IsOnStack(instance) || (allowStackFunction && !TaggedNumber::Is(instance) && (\*(int\*)instance & 1)))

{

return instance;

}

TypeId typeId = JavascriptOperators::GetTypeId(instance);

switch (typeId)

{

case Js::TypeIds\_Number:

#if !FLOATVAR

return JavascriptNumber::BoxStackInstance(instance, scriptContext);

#endif

// fall-through

case Js::TypeIds\_Integer:

return instance;

case Js::TypeIds\_RegEx:

return JavascriptRegExp::BoxStackInstance(JavascriptRegExp::FromVar(instance));

case Js::TypeIds\_Object:

return DynamicObject::BoxStackInstance(DynamicObject::FromVar(instance));

case Js::TypeIds\_Array:

return JavascriptArray::BoxStackInstance(JavascriptArray::FromVar(instance));

case Js::TypeIds\_NativeIntArray:

return JavascriptNativeIntArray::BoxStackInstance(JavascriptNativeIntArray::FromVar(instance));

case Js::TypeIds\_NativeFloatArray:

return JavascriptNativeFloatArray::BoxStackInstance(JavascriptNativeFloatArray::FromVar(instance));

case Js::TypeIds\_Function:

Assert(allowStackFunction);

// Stack functions are deal with not mar mark them, but by nested function escape analysis

// in the front end. No need to box here.

return instance;

#if ENABLE\_COPYONACCESS\_ARRAY

case Js::TypeIds\_CopyOnAccessNativeIntArray:

Assert(false);

// fall-through

#endif

default:

Assert(false);

return instance;

};

}

ImplicitCallFlags

JavascriptOperators::CacheAndClearImplicitBit(ScriptContext\* scriptContext)

{

ImplicitCallFlags prevImplicitCallFlags = scriptContext->GetThreadContext()->GetImplicitCallFlags();

scriptContext->GetThreadContext()->ClearImplicitCallFlags();

return prevImplicitCallFlags;

}

ImplicitCallFlags

JavascriptOperators::CheckAndUpdateFunctionBodyWithImplicitFlag(FunctionBody\* functionBody)

{

ScriptContext\* scriptContext = functionBody->GetScriptContext();

ImplicitCallFlags currImplicitCallFlags = scriptContext->GetThreadContext()->GetImplicitCallFlags();

if ((currImplicitCallFlags > ImplicitCall\_None))

{

functionBody->SetHasOnlyThisStmts(false);

}

return currImplicitCallFlags;

}

void

JavascriptOperators::RestoreImplicitFlag(ScriptContext\* scriptContext, ImplicitCallFlags prevImplicitCallFlags, ImplicitCallFlags currImplicitCallFlags)

{

scriptContext->GetThreadContext()->SetImplicitCallFlags((ImplicitCallFlags)(prevImplicitCallFlags | currImplicitCallFlags));

}

FunctionProxy\*

JavascriptOperators::GetDeferredDeserializedFunctionProxy(JavascriptFunction\* func)

{

FunctionProxy\* proxy = func->GetFunctionProxy();

if (proxy->GetFunctionProxy() != proxy)

{

proxy = proxy->GetFunctionProxy();

}

return proxy;

}

template <>

Js::Var JavascriptOperators::GetElementAtIndex(Js::JavascriptArray\* arrayObject, UINT index, Js::ScriptContext\* scriptContext)

{

Js::Var result;

if (Js::JavascriptOperators::OP\_GetElementI\_ArrayFastPath(arrayObject, index, &result, scriptContext))

{

return result;

}

return scriptContext->GetMissingItemResult(arrayObject, index);

}

template<>

Js::Var JavascriptOperators::GetElementAtIndex(Js::JavascriptNativeIntArray\* arrayObject, UINT index, Js::ScriptContext\* scriptContext)

{

Js::Var result;

if (Js::JavascriptOperators::OP\_GetElementI\_ArrayFastPath(arrayObject, index, &result, scriptContext))

{

return result;

}

return scriptContext->GetMissingItemResult(arrayObject, index);

}

template<>

Js::Var JavascriptOperators::GetElementAtIndex(Js::JavascriptNativeFloatArray\* arrayObject, UINT index, Js::ScriptContext\* scriptContext)

{

Js::Var result;

if (Js::JavascriptOperators::OP\_GetElementI\_ArrayFastPath(arrayObject, index, &result, scriptContext))

{

return result;

}

return scriptContext->GetMissingItemResult(arrayObject, index);

}

template<>

Js::Var JavascriptOperators::GetElementAtIndex(Js::Var\* arrayObject, UINT index, Js::ScriptContext\* scriptContext)

{

return Js::JavascriptOperators::OP\_GetElementI\_Int32(\*arrayObject, index, scriptContext);

}

template<typename T>

void JavascriptOperators::ObjectToNativeArray(T\* arrayObject,

JsNativeValueType valueType,

\_\_in UINT length,

\_\_in UINT elementSize,

\_\_out\_bcount(length\*elementSize) byte\* buffer,

Js::ScriptContext\* scriptContext)

{

Var element;

uint64 allocSize = length \* elementSize;

// TODO:further fast path the call for things like IntArray convert to int, floatarray convert to float etc.

// such that we don't need boxing.

switch (valueType)

{

case JsInt8Type:

AnalysisAssert(elementSize == sizeof(int8));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(int8) <= allocSize);

((int8\*)buffer)[i] = Js::JavascriptConversion::ToInt8(element, scriptContext);

}

break;

case JsUint8Type:

AnalysisAssert(elementSize == sizeof(uint8));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(uint8) <= allocSize);

((uint8\*)buffer)[i] = Js::JavascriptConversion::ToUInt8(element, scriptContext);

}

break;

case JsInt16Type:

AnalysisAssert(elementSize == sizeof(int16));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(int16) <= allocSize);

((int16\*)buffer)[i] = Js::JavascriptConversion::ToInt16(element, scriptContext);

}

break;

case JsUint16Type:

AnalysisAssert(elementSize == sizeof(uint16));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(uint16) <= allocSize);

((uint16\*)buffer)[i] = Js::JavascriptConversion::ToUInt16(element, scriptContext);

}

break;

case JsInt32Type:

AnalysisAssert(elementSize == sizeof(int32));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(int32) <= allocSize);

((int32\*)buffer)[i] = Js::JavascriptConversion::ToInt32(element, scriptContext);

}

break;

case JsUint32Type:

AnalysisAssert(elementSize == sizeof(uint32));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(uint32) <= allocSize);

((uint32\*)buffer)[i] = Js::JavascriptConversion::ToUInt32(element, scriptContext);

}

break;

case JsInt64Type:

AnalysisAssert(elementSize == sizeof(int64));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(int64) <= allocSize);

((int64\*)buffer)[i] = Js::JavascriptConversion::ToInt64(element, scriptContext);

}

break;

case JsUint64Type:

AnalysisAssert(elementSize == sizeof(uint64));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(uint64) <= allocSize);

((uint64\*)buffer)[i] = Js::JavascriptConversion::ToUInt64(element, scriptContext);

}

break;

case JsFloatType:

AnalysisAssert(elementSize == sizeof(float));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(float) <= allocSize);

((float\*)buffer)[i] = Js::JavascriptConversion::ToFloat(element, scriptContext);

}

break;

case JsDoubleType:

AnalysisAssert(elementSize == sizeof(double));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(double) <= allocSize);

((double\*)buffer)[i] = Js::JavascriptConversion::ToNumber(element, scriptContext);

}

break;

case JsNativeStringType:

AnalysisAssert(elementSize == sizeof(JsNativeString));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(JsNativeString) <= allocSize);

Js::JavascriptString\* string = Js::JavascriptConversion::ToString(element, scriptContext);

(((JsNativeString\*)buffer)[i]).str = string->GetSz();

(((JsNativeString\*)buffer)[i]).length = string->GetLength();

}

break;

default:

Assert(FALSE);

}

}

void JavascriptOperators::VarToNativeArray(Var arrayObject,

JsNativeValueType valueType,

\_\_in UINT length,

\_\_in UINT elementSize,

\_\_out\_bcount(length\*elementSize) byte\* buffer,

Js::ScriptContext\* scriptContext)

{

Js::DynamicObject\* dynamicObject = DynamicObject::FromVar(arrayObject);

if (dynamicObject->IsCrossSiteObject() || Js::TaggedInt::IsOverflow(length))

{

Js::JavascriptOperators::ObjectToNativeArray(&arrayObject, valueType, length, elementSize, buffer, scriptContext);

}

else

{

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(arrayObject);

#endif

switch (Js::JavascriptOperators::GetTypeId(arrayObject))

{

case TypeIds\_Array:

Js::JavascriptOperators::ObjectToNativeArray(Js::JavascriptArray::FromVar(arrayObject), valueType, length, elementSize, buffer, scriptContext);

break;

case TypeIds\_NativeFloatArray:

Js::JavascriptOperators::ObjectToNativeArray(Js::JavascriptNativeFloatArray::FromVar(arrayObject), valueType, length, elementSize, buffer, scriptContext);

break;

case TypeIds\_NativeIntArray:

Js::JavascriptOperators::ObjectToNativeArray(Js::JavascriptNativeIntArray::FromVar(arrayObject), valueType, length, elementSize, buffer, scriptContext);

break;

// We can have more specialized template if needed.

default:

Js::JavascriptOperators::ObjectToNativeArray(&arrayObject, valueType, length, elementSize, buffer, scriptContext);

}

}

}

// SpeciesConstructor abstract operation as described in ES6.0 Section 7.3.20

Var JavascriptOperators::SpeciesConstructor(RecyclableObject\* object, Var defaultConstructor, ScriptContext\* scriptContext)

{

//1.Assert: Type(O) is Object.

Assert(JavascriptOperators::IsObject(object));

//2.Let C be Get(O, "constructor").

//3.ReturnIfAbrupt(C).

Var constructor = JavascriptOperators::GetProperty(object, PropertyIds::constructor, scriptContext);

if (scriptContext->GetConfig()->IsES6SpeciesEnabled())

{

//4.If C is undefined, return defaultConstructor.

if (JavascriptOperators::IsUndefinedObject(constructor))

{

return defaultConstructor;

}

//5.If Type(C) is not Object, throw a TypeError exception.

if (!JavascriptOperators::IsObject(constructor))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedObject, L"[constructor]");

}

//6.Let S be Get(C, @@species).

//7.ReturnIfAbrupt(S).

Var species = nullptr;

if (!JavascriptOperators::GetProperty(RecyclableObject::FromVar(constructor), PropertyIds::\_symbolSpecies, &species, scriptContext)

|| JavascriptOperators::IsUndefinedOrNullType(JavascriptOperators::GetTypeId(species)))

{

//8.If S is either undefined or null, return defaultConstructor.

return defaultConstructor;

}

constructor = species;

}

//9.If IsConstructor(S) is true, return S.

if (JavascriptOperators::IsConstructor(constructor))

{

return constructor;

}

//10.Throw a TypeError exception.

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NotAConstructor, L"constructor[Symbol.species]");

}

BOOL JavascriptOperators::GreaterEqual(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

if (TaggedInt::Is(aLeft))

{

if (TaggedInt::Is(aRight))

{

// Works whether it is TaggedInt31 or TaggedInt32

return ::Math::PointerCastToIntegralTruncate<int>(aLeft) >= ::Math::PointerCastToIntegralTruncate<int>(aRight);

}

if (JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return TaggedInt::ToDouble(aLeft) >= JavascriptNumber::GetValue(aRight);

}

}

else if (TaggedInt::Is(aRight))

{

if (JavascriptNumber::Is\_NoTaggedIntCheck(aLeft))

{

return JavascriptNumber::GetValue(aLeft) >= TaggedInt::ToDouble(aRight);

}

}

else

{

if (JavascriptNumber::Is\_NoTaggedIntCheck(aLeft) && JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return JavascriptNumber::GetValue(aLeft) >= JavascriptNumber::GetValue(aRight);

}

}

return !RelationalComparsionHelper(aLeft, aRight, scriptContext, true, true);

}

BOOL JavascriptOperators::LessEqual(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

if (TaggedInt::Is(aLeft))

{

if (TaggedInt::Is(aRight))

{

// Works whether it is TaggedInt31 or TaggedInt32

return ::Math::PointerCastToIntegralTruncate<int>(aLeft) <= ::Math::PointerCastToIntegralTruncate<int>(aRight);

}

if (JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return TaggedInt::ToDouble(aLeft) <= JavascriptNumber::GetValue(aRight);

}

}

else if (TaggedInt::Is(aRight))

{

if (JavascriptNumber::Is\_NoTaggedIntCheck(aLeft))

{

return JavascriptNumber::GetValue(aLeft) <= TaggedInt::ToDouble(aRight);

}

}

else

{

if (JavascriptNumber::Is\_NoTaggedIntCheck(aLeft) && JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return JavascriptNumber::GetValue(aLeft) <= JavascriptNumber::GetValue(aRight);

}

}

return !RelationalComparsionHelper(aRight, aLeft, scriptContext, false, true);

}

BOOL JavascriptOperators::NotEqual(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

//

// TODO: Change to use Abstract Equality Comparison Algorithm (ES3.0: S11.9.3):

// - Evaluate left, then right, operands to preserve correct evaluation order.

// - Call algorithm, potentially reversing arguments.

//

return !Equal(aLeft, aRight, scriptContext);

}

// NotStrictEqual() returns whether the two vars have strict equality, as

// described in (ES3.0: S11.9.5, S11.9.6).

BOOL JavascriptOperators::NotStrictEqual(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

return !StrictEqual(aLeft, aRight, scriptContext);

}

bool JavascriptOperators::CheckIfObjectAndPrototypeChainHasOnlyWritableDataProperties(RecyclableObject\* object)

{

Assert(object);

if (object->GetType()->HasSpecialPrototype())

{

TypeId typeId = object->GetTypeId();

if (typeId == TypeIds\_Null)

{

return true;

}

if (typeId == TypeIds\_Proxy)

{

return false;

}

}

if (!object->HasOnlyWritableDataProperties())

{

return false;

}

return CheckIfPrototypeChainHasOnlyWritableDataProperties(object->GetPrototype());

}

bool JavascriptOperators::CheckIfPrototypeChainHasOnlyWritableDataProperties(RecyclableObject\* prototype)

{

Assert(prototype);

if (prototype->GetType()->AreThisAndPrototypesEnsuredToHaveOnlyWritableDataProperties())

{

Assert(DoCheckIfPrototypeChainHasOnlyWritableDataProperties(prototype));

return true;

}

return DoCheckIfPrototypeChainHasOnlyWritableDataProperties(prototype);

}

// Does a quick check to see if the specified object (which should be a prototype object) and all objects in its prototype

// chain have only writable data properties (i.e. no accessors or non-writable properties).

bool JavascriptOperators::DoCheckIfPrototypeChainHasOnlyWritableDataProperties(RecyclableObject\* prototype)

{

Assert(prototype);

Type \*const originalType = prototype->GetType();

ScriptContext \*const scriptContext = prototype->GetScriptContext();

bool onlyOneScriptContext = true;

TypeId typeId;

for (; (typeId = prototype->GetTypeId()) != TypeIds\_Null; prototype = prototype->GetPrototype())

{

if (typeId == TypeIds\_Proxy)

{

return false;

}

if (!prototype->HasOnlyWritableDataProperties())

{

return false;

}

if (prototype->GetScriptContext() != scriptContext)

{

onlyOneScriptContext = false;

}

}

if (onlyOneScriptContext)

{

// See JavascriptLibrary::typesEnsuredToHaveOnlyWritableDataPropertiesInItAndPrototypeChain for a description of

// this cache. Technically, we could register all prototypes in the chain but this is good enough for now.

originalType->SetAreThisAndPrototypesEnsuredToHaveOnlyWritableDataProperties(true);

}

return true;

}

BOOL JavascriptOperators::Equal(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

if (aLeft == aRight)

{

if (TaggedInt::Is(aLeft) || JavascriptObject::Is(aLeft))

{

return true;

}

else

{

return Equal\_Full(aLeft, aRight, scriptContext);

}

}

if (JavascriptString::Is(aLeft) && JavascriptString::Is(aRight))

{

JavascriptString\* left = (JavascriptString\*)aLeft;

JavascriptString\* right = (JavascriptString\*)aRight;

if (left->GetLength() == right->GetLength())

{

if (left->UnsafeGetBuffer() != NULL && right->UnsafeGetBuffer() != NULL)

{

if (left->GetLength() == 1)

{

return left->UnsafeGetBuffer()[0] == right->UnsafeGetBuffer()[0];

}

return memcmp(left->UnsafeGetBuffer(), right->UnsafeGetBuffer(), left->GetLength() \* sizeof(left->UnsafeGetBuffer()[0])) == 0;

}

// fall through to Equal\_Full

}

else

{

return false;

}

}

return Equal\_Full(aLeft, aRight, scriptContext);

}

BOOL JavascriptOperators::Greater(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

if (TaggedInt::Is(aLeft))

{

if (TaggedInt::Is(aRight))

{

// Works whether it is TaggedInt31 or TaggedInt32

return ::Math::PointerCastToIntegralTruncate<int>(aLeft) > ::Math::PointerCastToIntegralTruncate<int>(aRight);

}

if (JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return TaggedInt::ToDouble(aLeft) > JavascriptNumber::GetValue(aRight);

}

}

else if (TaggedInt::Is(aRight))

{

if (JavascriptNumber::Is\_NoTaggedIntCheck(aLeft))

{

return JavascriptNumber::GetValue(aLeft) > TaggedInt::ToDouble(aRight);

}

}

else

{

if (JavascriptNumber::Is\_NoTaggedIntCheck(aLeft) && JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return JavascriptNumber::GetValue(aLeft) > JavascriptNumber::GetValue(aRight);

}

}

return Greater\_Full(aLeft, aRight, scriptContext);

}

BOOL JavascriptOperators::Less(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

if (TaggedInt::Is(aLeft))

{

if (TaggedInt::Is(aRight))

{

// Works whether it is TaggedInt31 or TaggedInt32

return ::Math::PointerCastToIntegralTruncate<int>(aLeft) < ::Math::PointerCastToIntegralTruncate<int>(aRight);

}

if (JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return TaggedInt::ToDouble(aLeft) < JavascriptNumber::GetValue(aRight);

}

}

else if (TaggedInt::Is(aRight))

{

if (JavascriptNumber::Is\_NoTaggedIntCheck(aLeft))

{

return JavascriptNumber::GetValue(aLeft) < TaggedInt::ToDouble(aRight);

}

}

else

{

if (JavascriptNumber::Is\_NoTaggedIntCheck(aLeft) && JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return JavascriptNumber::GetValue(aLeft) < JavascriptNumber::GetValue(aRight);

}

}

return Less\_Full(aLeft, aRight, scriptContext);

}

Var JavascriptOperators::ToObject(Var aRight, ScriptContext\* scriptContext)

{

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptConversion::ToObject(aRight, scriptContext, &object))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedObject /\* TODO-ERROR: get arg name - aValue \*/);

}

return object;

}

Var JavascriptOperators::ToWithObject(Var aRight, ScriptContext\* scriptContext)

{

RecyclableObject\* object = RecyclableObject::FromVar(aRight);

WithScopeObject\* withWrapper = RecyclerNew(scriptContext->GetRecycler(), WithScopeObject, object, scriptContext->GetLibrary()->GetWithType());

return withWrapper;

}

Var JavascriptOperators::ToNumber(Var aRight, ScriptContext\* scriptContext)

{

if (TaggedInt::Is(aRight) || (JavascriptNumber::Is\_NoTaggedIntCheck(aRight)))

{

return aRight;

}

return JavascriptNumber::ToVarNoCheck(JavascriptConversion::ToNumber\_Full(aRight, scriptContext), scriptContext);

}

BOOL JavascriptOperators::IsObject(Var aValue)

{

return GetTypeId(aValue) > TypeIds\_LastJavascriptPrimitiveType;

}

BOOL JavascriptOperators::IsObjectType(TypeId typeId)

{

return typeId > TypeIds\_LastJavascriptPrimitiveType;

}

BOOL JavascriptOperators::IsExposedType(TypeId typeId)

{

return typeId <= TypeIds\_LastTrueJavascriptObjectType && typeId != TypeIds\_HostDispatch;

}

BOOL JavascriptOperators::IsObjectOrNull(Var instance)

{

TypeId typeId = GetTypeId(instance);

return IsObjectType(typeId) || typeId == TypeIds\_Null;

}

BOOL JavascriptOperators::IsUndefinedOrNullType(TypeId typeId)

{

return typeId <= TypeIds\_UndefinedOrNull;

}

BOOL JavascriptOperators::IsSpecialObjectType(TypeId typeId)

{

return typeId > TypeIds\_LastTrueJavascriptObjectType;

}

BOOL JavascriptOperators::IsUndefinedObject(Var instance)

{

return JavascriptOperators::GetTypeId(instance) == TypeIds\_Undefined;

}

BOOL JavascriptOperators::IsUndefinedObject(Var instance, RecyclableObject \*libraryUndefined)

{

Assert(JavascriptOperators::IsUndefinedObject(libraryUndefined));

return instance == libraryUndefined;

}

BOOL JavascriptOperators::IsUndefinedObject(Var instance, ScriptContext \*scriptContext)

{

return JavascriptOperators::IsUndefinedObject(instance, scriptContext->GetLibrary()->GetUndefined());

}

BOOL JavascriptOperators::IsUndefinedObject(Var instance, JavascriptLibrary\* library)

{

return JavascriptOperators::IsUndefinedObject(instance, library->GetUndefined());

}

BOOL JavascriptOperators::IsAnyNumberValue(Var instance)

{

TypeId typeId = GetTypeId(instance);

return TypeIds\_FirstNumberType <= typeId && typeId <= TypeIds\_LastNumberType;

}

BOOL JavascriptOperators::IsIterable(RecyclableObject\* instance, ScriptContext\* scriptContext)

{

if (JavascriptProxy::Is(instance))

{

Var func = JavascriptOperators::GetProperty(instance, PropertyIds::\_symbolIterator, scriptContext);

if (JavascriptOperators::IsUndefinedObject(func))

{

return FALSE;

}

else

{

return TRUE;

}

}

else

{

return JavascriptOperators::HasProperty(instance, PropertyIds::\_symbolIterator);

}

}

// GetIterator as described in ES6.0 (draft 22) Section 7.4.1

RecyclableObject\* JavascriptOperators::GetIterator(Var iterable, ScriptContext\* scriptContext)

{

RecyclableObject\* iterableObj = RecyclableObject::FromVar(JavascriptOperators::ToObject(iterable, scriptContext));

return JavascriptOperators::GetIterator(iterableObj, scriptContext);

}

RecyclableObject\* JavascriptOperators::GetIteratorFunction(Var iterable, ScriptContext\* scriptContext)

{

RecyclableObject\* iterableObj = RecyclableObject::FromVar(JavascriptOperators::ToObject(iterable, scriptContext));

return JavascriptOperators::GetIteratorFunction(iterableObj, scriptContext);

}

RecyclableObject\* JavascriptOperators::GetIteratorFunction(RecyclableObject\* instance, ScriptContext \* scriptContext)

{

Var func = JavascriptOperators::GetProperty(instance, PropertyIds::\_symbolIterator, scriptContext);

if (!JavascriptConversion::IsCallable(func))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedFunction);

}

RecyclableObject\* function = RecyclableObject::FromVar(func);

return function;

}

RecyclableObject\* JavascriptOperators::GetIterator(RecyclableObject\* instance, ScriptContext \* scriptContext)

{

RecyclableObject\* function = GetIteratorFunction(instance, scriptContext);

Var iterator = function->GetEntryPoint()(function, CallInfo(Js::CallFlags\_Value, 1), instance);

if (!JavascriptOperators::IsObject(iterator))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedObject);

}

return RecyclableObject::FromVar(iterator);

}

// IteratorNext as described in ES6.0 (draft 22) Section 7.4.2

RecyclableObject\* JavascriptOperators::IteratorNext(RecyclableObject\* iterator, ScriptContext\* scriptContext, Var value)

{

Var func = JavascriptOperators::GetProperty(iterator, PropertyIds::next, scriptContext);

if (!JavascriptConversion::IsCallable(func))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedFunction);

}

RecyclableObject\* callable = RecyclableObject::FromVar(func);

Js::Var args[] = { iterator, value };

Js::CallInfo callInfo(Js::CallFlags\_Value, \_countof(args) + (value == nullptr ? -1 : 0));

Var result = JavascriptFunction::CallFunction<true>(callable, callable->GetEntryPoint(), Js::Arguments(callInfo, args));

if (!JavascriptOperators::IsObject(result))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedObject);

}

return RecyclableObject::FromVar(result);

}

// IteratorComplete as described in ES6.0 (draft 22) Section 7.4.3

bool JavascriptOperators::IteratorComplete(RecyclableObject\* iterResult, ScriptContext\* scriptContext)

{

Var done = JavascriptOperators::GetProperty(iterResult, Js::PropertyIds::done, scriptContext);

return JavascriptConversion::ToBool(done, scriptContext);

}

// IteratorValue as described in ES6.0 (draft 22) Section 7.4.4

Var JavascriptOperators::IteratorValue(RecyclableObject\* iterResult, ScriptContext\* scriptContext)

{

return JavascriptOperators::GetProperty(iterResult, Js::PropertyIds::value, scriptContext);

}

// IteratorStep as described in ES6.0 (draft 22) Section 7.4.5

bool JavascriptOperators::IteratorStep(RecyclableObject\* iterator, ScriptContext\* scriptContext, RecyclableObject\*\* result)

{

Assert(result);

\*result = JavascriptOperators::IteratorNext(iterator, scriptContext);

return !JavascriptOperators::IteratorComplete(\*result, scriptContext);

}

bool JavascriptOperators::IteratorStepAndValue(RecyclableObject\* iterator, ScriptContext\* scriptContext, Var\* resultValue)

{

RecyclableObject\* result = JavascriptOperators::IteratorNext(iterator, scriptContext);

if (!JavascriptOperators::IteratorComplete(result, scriptContext))

{

\*resultValue = JavascriptOperators::IteratorValue(result, scriptContext);

return true;

}

return false;

}

RecyclableObject\* JavascriptOperators::CreateFromConstructor(RecyclableObject\* constructor, ScriptContext\* scriptContext)

{

// Create a regular object and set the internal proto from the constructor

return JavascriptOperators::OrdinaryCreateFromConstructor(constructor, scriptContext->GetLibrary()->CreateObject(), nullptr, scriptContext);

}

RecyclableObject\* JavascriptOperators::OrdinaryCreateFromConstructor(RecyclableObject\* constructor, RecyclableObject\* obj, DynamicObject\* intrinsicProto, ScriptContext\* scriptContext)

{

// There isn't a good way for us to add internal properties to objects in Chakra.

// Thus, caller should take care to create obj with the correct internal properties.

Var proto = JavascriptOperators::GetProperty(constructor, Js::PropertyIds::prototype, scriptContext);

// If constructor.prototype is an object, we should use that as the [[Prototype]] for our obj.

// Else, we set the [[Prototype]] internal slot of obj to %intrinsicProto% - which should be the default.

if (JavascriptOperators::IsObjectType(JavascriptOperators::GetTypeId(proto)) &&

DynamicObject::FromVar(proto) != intrinsicProto)

{

JavascriptObject::ChangePrototype(obj, RecyclableObject::FromVar(proto), /\*validate\*/true, scriptContext);

}

return obj;

}

Var JavascriptOperators::GetProperty(RecyclableObject\* instance, PropertyId propertyId, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

return JavascriptOperators::GetProperty(instance, instance, propertyId, requestContext, info);

}

BOOL JavascriptOperators::GetProperty(RecyclableObject\* instance, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

return JavascriptOperators::GetProperty(instance, instance, propertyId, value, requestContext, info);

}

Var JavascriptOperators::GetProperty(Var instance, RecyclableObject\* propertyObject, PropertyId propertyId, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

Var value;

if (JavascriptOperators::GetProperty(instance, propertyObject, propertyId, &value, requestContext, info))

{

return value;

}

return requestContext->GetMissingPropertyResult(propertyObject, propertyId);

}

Var JavascriptOperators::GetRootProperty(RecyclableObject\* instance, PropertyId propertyId, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

Var value;

if (JavascriptOperators::GetRootProperty(instance, propertyId, &value, requestContext, info))

{

return value;

}

return requestContext->GetMissingPropertyResult(instance, propertyId);

}

BOOL JavascriptOperators::GetPropertyReference(RecyclableObject \*instance, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

return JavascriptOperators::GetPropertyReference(instance, instance, propertyId, value, requestContext, info);

}

BOOL JavascriptOperators::GetItem(RecyclableObject\* instance, uint64 index, Var\* value, ScriptContext\* requestContext)

{

PropertyRecord const \* propertyRecord;

JavascriptOperators::GetPropertyIdForInt(index, requestContext, &propertyRecord);

return JavascriptOperators::GetProperty(instance, propertyRecord->GetPropertyId(), value, requestContext);

}

BOOL JavascriptOperators::GetItem(RecyclableObject\* instance, uint32 index, Var\* value, ScriptContext\* requestContext)

{

return JavascriptOperators::GetItem(instance, instance, index, value, requestContext);

}

BOOL JavascriptOperators::GetItemReference(RecyclableObject\* instance, uint32 index, Var\* value, ScriptContext\* requestContext)

{

return GetItemReference(instance, instance, index, value, requestContext);

}

BOOL JavascriptOperators::CheckPrototypesForAccessorOrNonWritableProperty(RecyclableObject\* instance, PropertyId propertyId, Var\* setterValue, DescriptorFlags\* flags, PropertyValueInfo\* info, ScriptContext\* scriptContext)

{

if (propertyId == Js::PropertyIds::\_\_proto\_\_)

{

return CheckPrototypesForAccessorOrNonWritablePropertyCore<PropertyId, false, false>(instance, propertyId, setterValue, flags, info, scriptContext);

}

else

{

return CheckPrototypesForAccessorOrNonWritablePropertyCore<PropertyId, true, false>(instance, propertyId, setterValue, flags, info, scriptContext);

}

}

BOOL JavascriptOperators::CheckPrototypesForAccessorOrNonWritableRootProperty(RecyclableObject\* instance, PropertyId propertyId, Var\* setterValue, DescriptorFlags\* flags, PropertyValueInfo\* info, ScriptContext\* scriptContext)

{

if (propertyId == Js::PropertyIds::\_\_proto\_\_)

{

return CheckPrototypesForAccessorOrNonWritablePropertyCore<PropertyId, false, true>(instance, propertyId, setterValue, flags, info, scriptContext);

}

else

{

return CheckPrototypesForAccessorOrNonWritablePropertyCore<PropertyId, true, true>(instance, propertyId, setterValue, flags, info, scriptContext);

}

}

BOOL JavascriptOperators::CheckPrototypesForAccessorOrNonWritableProperty(RecyclableObject\* instance, JavascriptString\* propertyNameString, Var\* setterValue, DescriptorFlags\* flags, PropertyValueInfo\* info, ScriptContext\* scriptContext)

{

JsUtil::CharacterBuffer<WCHAR> propertyName(propertyNameString->GetString(), propertyNameString->GetLength());

if (Js::BuiltInPropertyRecords::\_\_proto\_\_.Equals(propertyName))

{

return CheckPrototypesForAccessorOrNonWritablePropertyCore<JavascriptString\*, false, false>(instance, propertyNameString, setterValue, flags, info, scriptContext);

}

else

{

return CheckPrototypesForAccessorOrNonWritablePropertyCore<JavascriptString\*, true, false>(instance, propertyNameString, setterValue, flags, info, scriptContext);

}

}

template<typename PropertyKeyType>

BOOL JavascriptOperators::CheckPrototypesForAccessorOrNonWritablePropertySlow(RecyclableObject\* instance, PropertyKeyType propertyKey, Var\* setterValue, DescriptorFlags\* flags, bool isRoot, ScriptContext\* scriptContext)

{

// This is used in debug verification, do not doFastProtoChainCheck to avoid side effect (doFastProtoChainCheck may update HasWritableDataOnly flags).

if (isRoot)

{

return CheckPrototypesForAccessorOrNonWritablePropertyCore<PropertyKeyType, /\*doFastProtoChainCheck\*/false, true>(instance, propertyKey, setterValue, flags, nullptr, scriptContext);

}

else

{

return CheckPrototypesForAccessorOrNonWritablePropertyCore<PropertyKeyType, /\*doFastProtoChainCheck\*/false, false>(instance, propertyKey, setterValue, flags, nullptr, scriptContext);

}

}

BOOL JavascriptOperators::SetProperty(Var instance, RecyclableObject\* object, PropertyId propertyId, Var newValue, ScriptContext\* requestContext, PropertyOperationFlags propertyOperationFlags)

{

PropertyValueInfo info;

return JavascriptOperators::SetProperty(instance, object, propertyId, newValue, &info, requestContext, propertyOperationFlags);

}

BOOL JavascriptOperators::TryConvertToUInt32(const wchar\_t\* str, int length, uint32\* intVal)

{

return NumberUtilities::TryConvertToUInt32(str, length, intVal);

}

template <typename TPropertyKey>

DescriptorFlags JavascriptOperators::GetRootSetter(RecyclableObject\* instance, TPropertyKey propertyKey, Var \*setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

// This is provided only so that CheckPrototypesForAccessorOrNonWritablePropertyCore will compile.

// It will never be called.

Throw::FatalInternalError();

}

template <>

inline DescriptorFlags JavascriptOperators::GetRootSetter(RecyclableObject\* instance, PropertyId propertyId, Var \*setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

AssertMsg(JavascriptOperators::GetTypeId(instance) == TypeIds\_GlobalObject

|| JavascriptOperators::GetTypeId(instance) == TypeIds\_ModuleRoot,

"Root must be a global object!");

RootObjectBase\* rootObject = static\_cast<RootObjectBase\*>(instance);

return rootObject->GetRootSetter(propertyId, setterValue, info, requestContext);

}

// Helper to fetch @@species from a constructor object

Var JavascriptOperators::GetSpecies(RecyclableObject\* constructor, ScriptContext\* scriptContext)

{

if (scriptContext->GetConfig()->IsES6SpeciesEnabled())

{

Var species = nullptr;

// Let S be Get(C, @@species)

if (JavascriptOperators::GetProperty(constructor, PropertyIds::\_symbolSpecies, &species, scriptContext)

&& !JavascriptOperators::IsUndefinedOrNullType(JavascriptOperators::GetTypeId(species)))

{

// If S is neither undefined nor null, let C be S

return species;

}

}

return constructor;

}

} // namespace Js