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

// 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

struct CodeGenWorkItem;

class SourceContextInfo;

class FunctionBailOutRecord;

struct DeferredFunctionStub;

#ifdef DYNAMIC\_PROFILE\_MUTATOR

class DynamicProfileMutator;

class DynamicProfileMutatorImpl;

#endif

#define MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE 42 //11\*3+8+1

namespace Js

{

#pragma region Class Forward Declarations

class ByteCodeBufferReader;

class ByteCodeBufferBuilder;

class ByteCodeCache;

class ScopeInfo;

class SmallSpanSequence;

struct StatementLocation;

class SmallSpanSequenceIter;

struct StatementData;

struct PropertyIdOnRegSlotsContainer;

struct InlineCache;

struct PolymorphicInlineCache;

struct IsInstInlineCache;

class ScopeObjectChain;

class EntryPointInfo;

class FunctionProxy;

class ParseableFunctionInfo;

class FunctionBody;

class DebuggerScopeProperty;

class DebuggerScope;

class FunctionEntryPointInfo;

#ifndef TEMP\_DISABLE\_ASMJS

class AsmJsFunctionInfo;

class AmsJsModuleInfo;

#endif

class ArrayBuffer;

class FunctionCodeGenRuntimeData;

#pragma endregion

typedef JsUtil::BaseDictionary<Js::PropertyId, const Js::PropertyRecord\*, RecyclerNonLeafAllocator, PowerOf2SizePolicy, DefaultComparer, JsUtil::SimpleDictionaryEntry> PropertyRecordList;

typedef JsUtil::BaseHashSet<void\*, Recycler, PowerOf2SizePolicy> TypeRefSet;

// Definition of scopes such as With, Catch and Block which will be used further in the debugger for additional look-ups.

enum DiagExtraScopesType

{

DiagUnknownScope, // Unknown scope set when deserializing bytecode and the scope is not yet known.

DiagWithScope, // With scope.

DiagCatchScopeDirect, // Catch scope in regslot

DiagCatchScopeInSlot, // Catch scope in slot array

DiagCatchScopeInObject, // Catch scope in scope object

DiagBlockScopeDirect, // Block scope in regslot

DiagBlockScopeInSlot, // Block scope in slot array

DiagBlockScopeInObject, // Block scope in activation object

DiagBlockScopeRangeEnd, // Used to end a block scope range.

};

class PropertyGuard

{

friend class PropertyGuardValidator;

private:

intptr\_t value;

public:

static PropertyGuard\* New(Recycler\* recycler) { return RecyclerNewLeaf(recycler, Js::PropertyGuard); }

PropertyGuard() : value(1) {}

PropertyGuard(intptr\_t value) : value(value) { Assert(this->value != 0); }

inline static size\_t const GetSizeOfValue() { return sizeof(((PropertyGuard\*)0)->value); }

inline static size\_t const GetOffsetOfValue() { return offsetof(PropertyGuard, value); }

intptr\_t GetValue() const { return this->value; }

bool IsValid() { return this->value != 0; }

void SetValue(intptr\_t value) { Assert(value != 0); this->value = value; }

intptr\_t const\* GetAddressOfValue() { return &this->value; }

void Invalidate() { this->value = 0; }

};

class PropertyGuardValidator

{

// Required by EquivalentTypeGuard::SetType.

CompileAssert(offsetof(PropertyGuard, value) == 0);

CompileAssert(offsetof(ConstructorCache, guard.value) == offsetof(PropertyGuard, value));

};

class JitIndexedPropertyGuard : public Js::PropertyGuard

{

private:

int index;

public:

JitIndexedPropertyGuard(intptr\_t value, int index):

Js::PropertyGuard(value), index(index) {}

int GetIndex() const { return this->index; }

};

class JitTypePropertyGuard : public Js::JitIndexedPropertyGuard

{

public:

JitTypePropertyGuard(Js::Type\* type, int index):

JitIndexedPropertyGuard(reinterpret\_cast<intptr\_t>(type), index) {}

Js::Type\* GetType() const { return reinterpret\_cast<Js::Type\*>(this->GetValue()); }

};

struct TypeGuardTransferEntry

{

PropertyId propertyId;

JitIndexedPropertyGuard\* guards[0];

TypeGuardTransferEntry(): propertyId(Js::Constants::NoProperty) {}

};

class FakePropertyGuardWeakReference: public RecyclerWeakReference<Js::PropertyGuard>

{

public:

static FakePropertyGuardWeakReference\* New(Recycler\* recycler, Js::PropertyGuard\* guard)

{

Assert(guard != nullptr);

return RecyclerNewLeaf(recycler, Js::FakePropertyGuardWeakReference, guard);

}

FakePropertyGuardWeakReference(const Js::PropertyGuard\* guard)

{

this->strongRef = (char\*)guard;

this->strongRefHeapBlock = &CollectedRecyclerWeakRefHeapBlock::Instance;

}

void Zero()

{

Assert(this->strongRef != nullptr);

this->strongRef = nullptr;

}

};

struct CtorCacheGuardTransferEntry

{

PropertyId propertyId;

ConstructorCache\* caches[0];

CtorCacheGuardTransferEntry(): propertyId(Js::Constants::NoProperty) {}

};

#define EQUIVALENT\_TYPE\_CACHE\_SIZE (8)

struct EquivalentTypeCache

{

Js::Type\* types[EQUIVALENT\_TYPE\_CACHE\_SIZE];

PropertyGuard \*guard;

TypeEquivalenceRecord record;

uint nextEvictionVictim;

bool isLoadedFromProto;

bool hasFixedValue;

EquivalentTypeCache(): nextEvictionVictim(EQUIVALENT\_TYPE\_CACHE\_SIZE) {}

bool ClearUnusedTypes(Recycler \*recycler);

void SetGuard(PropertyGuard \*theGuard) { this->guard = theGuard; }

void SetIsLoadedFromProto() { this->isLoadedFromProto = true; }

bool IsLoadedFromProto() const { return this->isLoadedFromProto; }

void SetHasFixedValue() { this->hasFixedValue = true; }

bool HasFixedValue() const { return this->hasFixedValue; }

};

class JitEquivalentTypeGuard : public JitIndexedPropertyGuard

{

// This pointer is allocated from background thread first, and then transferred to recycler,

// so as to keep the cached types alive.

EquivalentTypeCache\* cache;

uint32 objTypeSpecFldId;

public:

JitEquivalentTypeGuard(Type\* type, int index, uint32 objTypeSpecFldId):

JitIndexedPropertyGuard(reinterpret\_cast<intptr\_t>(type), index), cache(nullptr), objTypeSpecFldId(objTypeSpecFldId) {}

Js::Type\* GetType() const { return reinterpret\_cast<Js::Type\*>(this->GetValue()); }

void SetType(const Js::Type\* type)

{

this->SetValue(reinterpret\_cast<intptr\_t>(type));

}

uint32 GetObjTypeSpecFldId() const

{

return this->objTypeSpecFldId;

}

Js::EquivalentTypeCache\* GetCache() const

{

return this->cache;

}

void SetCache(Js::EquivalentTypeCache\* cache)

{

this->cache = cache;

}

};

#pragma region Inline Cache Info class declarations

class PolymorphicCacheUtilizationArray

{

private:

byte \*utilArray;

public:

PolymorphicCacheUtilizationArray()

: utilArray(nullptr)

{

}

void EnsureUtilArray(Recycler \* const recycler, Js::FunctionBody \* functionBody);

void SetUtil(Js::FunctionBody\* functionBody, uint index, byte util);

byte GetUtil(Js::FunctionBody\* functionBody, uint index);

};

class PolymorphicInlineCacheInfo sealed

{

private:

InlineCachePointerArray<PolymorphicInlineCache> polymorphicInlineCaches;

PolymorphicCacheUtilizationArray polymorphicCacheUtilizationArray;

FunctionBody \* functionBody;

public:

PolymorphicInlineCacheInfo(FunctionBody \* functionBody)

: functionBody(functionBody)

{

}

InlineCachePointerArray<PolymorphicInlineCache> \* GetPolymorphicInlineCaches() { return &polymorphicInlineCaches; }

PolymorphicCacheUtilizationArray \* GetUtilArray() { return &polymorphicCacheUtilizationArray; }

FunctionBody \* GetFunctionBody() { return functionBody; }

};

class EntryPointPolymorphicInlineCacheInfo sealed

{

private:

PolymorphicInlineCacheInfo selfInfo;

SListBase<PolymorphicInlineCacheInfo\*> inlineeInfo;

static void SetPolymorphicInlineCache(PolymorphicInlineCacheInfo \* polymorphicInlineCacheInfo, FunctionBody \* functionBody, uint index, PolymorphicInlineCache \* polymorphicInlineCache, byte polyCacheUtil);

public:

EntryPointPolymorphicInlineCacheInfo(FunctionBody \* functionBody)

: selfInfo(functionBody)

{

}

PolymorphicInlineCacheInfo \* GetSelfInfo() { return &selfInfo; }

PolymorphicInlineCacheInfo \* EnsureInlineeInfo(Recycler \* recycler, FunctionBody \* inlineeFunctionBody);

PolymorphicInlineCacheInfo \* GetInlineeInfo(FunctionBody \* inlineeFunctionBody);

void SetPolymorphicInlineCache(FunctionBody \* functionBody, uint index, PolymorphicInlineCache \* polymorphicInlineCache, bool isInlinee, byte polyCacheUtil);

template <class Fn>

void MapInlinees(Fn fn)

{

SListBase<PolymorphicInlineCacheInfo\*>::Iterator iter(&inlineeInfo);

while (iter.Next())

{

fn(iter.Data());

}

}

};

#pragma endregion

#ifdef FIELD\_ACCESS\_STATS

struct FieldAccessStats

{

uint totalInlineCacheCount;

uint noInfoInlineCacheCount;

uint monoInlineCacheCount;

uint emptyMonoInlineCacheCount;

uint polyInlineCacheCount;

uint nullPolyInlineCacheCount;

uint emptyPolyInlineCacheCount;

uint ignoredPolyInlineCacheCount;

uint highUtilPolyInlineCacheCount;

uint lowUtilPolyInlineCacheCount;

uint equivPolyInlineCacheCount;

uint nonEquivPolyInlineCacheCount;

uint disabledPolyInlineCacheCount;

uint clonedMonoInlineCacheCount;

uint clonedPolyInlineCacheCount;

FieldAccessStats() :

totalInlineCacheCount(0), noInfoInlineCacheCount(0), monoInlineCacheCount(0), emptyMonoInlineCacheCount(0),

polyInlineCacheCount(0), nullPolyInlineCacheCount(0), emptyPolyInlineCacheCount(0), ignoredPolyInlineCacheCount(0),

highUtilPolyInlineCacheCount(0), lowUtilPolyInlineCacheCount(0),

equivPolyInlineCacheCount(0), nonEquivPolyInlineCacheCount(0), disabledPolyInlineCacheCount(0),

clonedMonoInlineCacheCount(0), clonedPolyInlineCacheCount(0) {}

void Add(FieldAccessStats\* other);

};

typedef FieldAccessStats\* FieldAccessStatsPtr;

#else

typedef void\* FieldAccessStatsPtr;

#endif

#pragma region Entry point class declarations

class ProxyEntryPointInfo: public ExpirableObject

{

public:

// These are public because we don't manage them nor their consistency;

// the user of this class does.

void \* address;

ProxyEntryPointInfo(void\* address, ThreadContext\* context = nullptr):

ExpirableObject(context),

address(address)

{

}

static DWORD GetAddressOffset() { return offsetof(ProxyEntryPointInfo, address); }

virtual void Expire()

{

AssertMsg(false, "Expire called on object that doesn't support expiration");

}

virtual void EnterExpirableCollectMode()

{

AssertMsg(false, "EnterExpirableCollectMode called on object that doesn't support expiration");

}

virtual bool IsFunctionEntryPointInfo() const { return false; }

};

// Not thread safe.

// Note that instances of this class are read from and written to from the

// main and JIT threads.

class EntryPointInfo : public ProxyEntryPointInfo

{

private:

enum State : BYTE

{

NotScheduled, // code gen has not been scheduled

CodeGenPending, // code gen job has been scheduled

CodeGenQueued, // code gen has been queued and all the code gen data has been gathered.

CodeGenRecorded, // backend completed, but job still pending

CodeGenDone, // code gen job successfully completed

JITCapReached, // workitem created but JIT cap reached

PendingCleanup, // workitem needs to be cleaned up but couldn't for some reason- it'll be cleaned up at the next opportunity

CleanedUp // the entry point has been cleaned up

};

#if ENABLE\_NATIVE\_CODEGEN

class JitTransferData

{

friend EntryPointInfo;

private:

TypeRefSet\* jitTimeTypeRefs;

void\*\* runtimeTypeRefs;

int runtimeTypeRefCount;

int propertyGuardCount;

// This is a dynamically sized array of dynamically sized TypeGuardTransferEntries. It's heap allocated by the JIT

// thread and lives until entry point is installed, at which point it is explicitly freed.

TypeGuardTransferEntry\* propertyGuardsByPropertyId;

size\_t propertyGuardsByPropertyIdPlusSize;

// This is a dynamically sized array of dynamically sized CtorCacheGuardTransferEntry. It's heap allocated by the JIT

// thread and lives until entry point is installed, at which point it is explicitly freed.

CtorCacheGuardTransferEntry\* ctorCacheGuardsByPropertyId;

size\_t ctorCacheGuardsByPropertyIdPlusSize;

int equivalentTypeGuardCount;

int lazyBailoutPropertyCount;

// This is a dynamically sized array of JitEquivalentTypeGuards. It's heap allocated by the JIT thread and lives

// until entry point is installed, at which point it is explicitly freed. We need it during installation so as to

// swap the cache associated with each guard from the heap to the recycler (so the types in the cache are kept alive).

JitEquivalentTypeGuard\*\* equivalentTypeGuards;

Js::PropertyId\* lazyBailoutProperties;

NativeCodeData\* data;

bool falseReferencePreventionBit;

bool isReady;

public:

JitTransferData():

jitTimeTypeRefs(nullptr), runtimeTypeRefCount(0), runtimeTypeRefs(nullptr),

propertyGuardCount(0), propertyGuardsByPropertyId(nullptr), propertyGuardsByPropertyIdPlusSize(0),

ctorCacheGuardsByPropertyId(nullptr), ctorCacheGuardsByPropertyIdPlusSize(0),

equivalentTypeGuardCount(0), equivalentTypeGuards(nullptr), data(nullptr),

falseReferencePreventionBit(true), isReady(false), lazyBailoutProperties(nullptr), lazyBailoutPropertyCount(0){}

void AddJitTimeTypeRef(void\* typeRef, Recycler\* recycler);

int GetRuntimeTypeRefCount() { return this->runtimeTypeRefCount; }

void\*\* GetRuntimeTypeRefs() { return this->runtimeTypeRefs; }

void SetRuntimeTypeRefs(void\*\* runtimeTypeRefs, int count) { this->runtimeTypeRefs = runtimeTypeRefs; this->runtimeTypeRefCount = count; }

JitEquivalentTypeGuard\*\* GetEquivalentTypeGuards() const { return this->equivalentTypeGuards; }

void SetEquivalentTypeGuards(JitEquivalentTypeGuard\*\* guards, int count)

{

this->equivalentTypeGuardCount = count;

this->equivalentTypeGuards = guards;

}

void SetLazyBailoutProperties(Js::PropertyId\* properties, int count)

{

this->lazyBailoutProperties = properties;

this->lazyBailoutPropertyCount = count;

}

bool GetIsReady() { return this->isReady; }

void SetIsReady() { this->isReady = true; }

private:

void EnsureJitTimeTypeRefs(Recycler\* recycler);

};

NativeCodeData \* data;

CodeGenNumberChunk \* numberChunks;

SmallSpanSequence \*nativeThrowSpanSequence;

typedef JsUtil::BaseHashSet<RecyclerWeakReference<FunctionBody>\*, Recycler, PowerOf2SizePolicy> WeakFuncRefSet;

WeakFuncRefSet \*weakFuncRefSet;

// Need to keep strong references to the guards here so they don't get collected while the entry point is alive.

typedef JsUtil::BaseDictionary<Js::PropertyId, PropertyGuard\*, Recycler, PowerOf2SizePolicy> SharedPropertyGuardDictionary;

SharedPropertyGuardDictionary\* sharedPropertyGuards;

typedef JsUtil::List<LazyBailOutRecord, HeapAllocator> BailOutRecordMap;

BailOutRecordMap\* bailoutRecordMap;

// This array holds fake weak references to type property guards. We need it to zero out the weak references when the

// entry point is finalized and the guards are about to be freed. Otherwise, if one of the guards was to be invalidated

// from the thread context, we would AV trying to access freed memory. Note that the guards themselves are allocated by

// NativeCodeData::Allocator and are kept alive by the data field. The weak references are recycler allocated, and so

// the array must be recycler allocated also, so that the recycler doesn't collect the weak references.

FakePropertyGuardWeakReference\*\* propertyGuardWeakRefs;

EquivalentTypeCache\* equivalentTypeCaches;

EntryPointInfo \*\* registeredEquivalentTypeCacheRef;

int propertyGuardCount;

int equivalentTypeCacheCount;

#endif

CodeGenWorkItem \* workItem;

void \* nativeAddress;

ptrdiff\_t codeSize;

bool isAsmJsFunction; // true if entrypoint is for asmjs function

uintptr mModuleAddress; //asm Module address

#ifdef FIELD\_ACCESS\_STATS

FieldAccessStatsPtr fieldAccessStats;

#endif

protected:

JavascriptLibrary\* library;

#if ENABLE\_NATIVE\_CODEGEN

typedef JsUtil::List<NativeOffsetInlineeFramePair, HeapAllocator> InlineeFrameMap;

InlineeFrameMap\* inlineeFrameMap;

#endif

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

StackBackTrace\* cleanupStack;

#endif

public:

enum CleanupReason

{

NotCleanedUp,

CodeGenFailedOOM,

CodeGenFailedStackOverflow,

CodeGenFailedAborted,

NativeCodeInstallFailure,

CleanUpForFinalize

};

uint frameHeight;

private:

#if ENABLE\_NATIVE\_CODEGEN

typedef SListCounted<ConstructorCache\*, Recycler> ConstructorCacheList;

ConstructorCacheList\* constructorCaches;

EntryPointPolymorphicInlineCacheInfo \* polymorphicInlineCacheInfo;

// This field holds any recycler allocated references that must be kept alive until

// we install the entry point. It is freed at that point, so anything that must survive

// until the EntryPointInfo itself goes away, must be copied somewhere else.

JitTransferData\* jitTransferData;

// If we pin types this array contains strong references to types, otherwise it holds weak references.

void \*\*runtimeTypeRefs;

uint32 pendingPolymorphicCacheState;

#endif

State state; // Single state member so users can query state w/o a lock

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

CleanupReason cleanupReason;

#endif

BYTE pendingInlinerVersion;

bool isLoopBody;

bool hasJittedStackClosure;

#if ENABLE\_NATIVE\_CODEGEN

ImplicitCallFlags pendingImplicitCallFlags;

#endif

public:

virtual void Finalize(bool isShutdown) override;

virtual bool IsFunctionEntryPointInfo() const override { return true; }

protected:

EntryPointInfo(void\* address, JavascriptLibrary\* library, void\* validationCookie, ThreadContext\* context = nullptr, bool isLoopBody = false) :

ProxyEntryPointInfo(address, context),

#if ENABLE\_NATIVE\_CODEGEN

nativeThrowSpanSequence(nullptr), workItem(nullptr), weakFuncRefSet(nullptr),

jitTransferData(nullptr), sharedPropertyGuards(nullptr), propertyGuardCount(0), propertyGuardWeakRefs(nullptr),

equivalentTypeCacheCount(0), equivalentTypeCaches(nullptr), constructorCaches(nullptr), state(NotScheduled), data(nullptr),

numberChunks(nullptr), polymorphicInlineCacheInfo(nullptr), runtimeTypeRefs(nullptr),

isLoopBody(isLoopBody), hasJittedStackClosure(false), registeredEquivalentTypeCacheRef(nullptr), bailoutRecordMap(nullptr),

#endif

library(library), codeSize(0), nativeAddress(nullptr), isAsmJsFunction(false), validationCookie(validationCookie)

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

, cleanupStack(nullptr)

, cleanupReason(NotCleanedUp)

#endif

#if DBG\_DUMP | defined(VTUNE\_PROFILING)

, nativeOffsetMaps(&HeapAllocator::Instance)

#endif

#ifdef FIELD\_ACCESS\_STATS

, fieldAccessStats(nullptr)

#endif

{}

virtual void ReleasePendingWorkItem() {};

virtual void OnCleanup(bool isShutdown) = 0;

#ifdef PERF\_COUNTERS

virtual void OnRecorded() = 0;

#endif

private:

State GetState() const

{

Assert(this->state >= NotScheduled && this->state <= CleanedUp);

return this->state;

}

public:

ScriptContext\* GetScriptContext();

virtual FunctionBody \*GetFunctionBody() const = 0;

#if ENABLE\_NATIVE\_CODEGEN

EntryPointPolymorphicInlineCacheInfo \* EnsurePolymorphicInlineCacheInfo(Recycler \* recycler, FunctionBody \* functionBody);

EntryPointPolymorphicInlineCacheInfo \* GetPolymorphicInlineCacheInfo() { return polymorphicInlineCacheInfo; }

JitTransferData\* GetJitTransferData() { return this->jitTransferData; }

JitTransferData\* EnsureJitTransferData(Recycler\* recycler);

#ifdef FIELD\_ACCESS\_STATS

FieldAccessStats\* GetFieldAccessStats() { return this->fieldAccessStats; }

FieldAccessStats\* EnsureFieldAccessStats(Recycler\* recycler);

#endif

void PinTypeRefs(ScriptContext\* scriptContext);

void InstallGuards(ScriptContext\* scriptContext);

#endif

void Cleanup(bool isShutdown, bool captureCleanupStack);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

void CaptureCleanupStackTrace();

#endif

bool IsNotScheduled() const

{

return this->GetState() == NotScheduled;

}

bool IsCodeGenPending() const

{

return this->GetState() == CodeGenPending;

}

bool IsNativeCode() const

{

#if ENABLE\_NATIVE\_CODEGEN

return this->GetState() == CodeGenRecorded ||

this->GetState() == CodeGenDone;

#else

return false;

#endif

}

bool IsCodeGenDone() const

{

#if ENABLE\_NATIVE\_CODEGEN

return this->GetState() == CodeGenDone;

#else

return false;

#endif

}

bool IsCodeGenQueued() const

{

#if ENABLE\_NATIVE\_CODEGEN

return this->GetState() == CodeGenQueued;

#else

return false;

#endif

}

bool IsJITCapReached() const

{

#if ENABLE\_NATIVE\_CODEGEN

return this->GetState() == JITCapReached;

#else

return false;

#endif

}

bool IsCleanedUp() const

{

return this->GetState() == CleanedUp;

}

bool IsPendingCleanup() const

{

return this->GetState() == PendingCleanup;

}

void SetPendingCleanup()

{

this->state = PendingCleanup;

}

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

void SetCleanupReason(CleanupReason reason)

{

this->cleanupReason = reason;

}

#endif

bool IsLoopBody() const

{

return this->isLoopBody;

}

#if ENABLE\_NATIVE\_CODEGEN

bool HasJittedStackClosure() const

{

return this->hasJittedStackClosure;

}

void SetHasJittedStackClosure()

{

this->hasJittedStackClosure = true;

}

#endif

#ifndef TEMP\_DISABLE\_ASMJS

void SetModuleAddress(uintptr moduleAddress)

{

Assert(this->GetIsAsmJSFunction());

Assert(moduleAddress);

mModuleAddress = moduleAddress;

}

uintptr GetModuleAddress()const

{

Assert(this->GetIsAsmJSFunction());

Assert(mModuleAddress); // module address should not be null

return mModuleAddress;

}

#endif

void Reset(bool resetStateToNotScheduled = true);

#if ENABLE\_NATIVE\_CODEGEN

void SetCodeGenPending(CodeGenWorkItem \* workItem)

{

Assert(this->GetState() == NotScheduled || this->GetState() == CleanedUp);

Assert(workItem != nullptr);

this->workItem = workItem;

this->state = CodeGenPending;

}

void SetCodeGenPending()

{

Assert(this->GetState() == CodeGenQueued);

this->state = CodeGenPending;

}

void SetCodeGenQueued()

{

Assert(this->GetState() == CodeGenPending);

this->state = CodeGenQueued;

}

void RevertToNotScheduled()

{

Assert(this->GetState() == CodeGenPending);

Assert(this->workItem != nullptr);

this->workItem = nullptr;

this->state = NotScheduled;

}

void SetCodeGenPendingWithStackAllocatedWorkItem()

{

Assert(this->GetState() == NotScheduled || this->GetState() == CleanedUp);

this->workItem = nullptr;

this->state = CodeGenPending;

}

void SetCodeGenRecorded(void \* nativeAddress, ptrdiff\_t codeSize,

NativeCodeData \* data, NativeCodeData \* transferData, CodeGenNumberChunk \* numberChunks)

{

Assert(this->GetState() == CodeGenQueued);

Assert(nativeAddress != nullptr);

Assert(codeSize > 0);

Assert(this->jitTransferData != nullptr || transferData == nullptr);

this->nativeAddress = (void \*)nativeAddress;

this->codeSize = codeSize;

this->data = data;

if (transferData != nullptr)

{

this->jitTransferData->data = transferData;

}

this->numberChunks = numberChunks;

this->state = CodeGenRecorded;

#ifdef PERF\_COUNTERS

this->OnRecorded();

#endif

}

void SetCodeGenDone()

{

Assert(this->GetState() == CodeGenRecorded);

this->state = CodeGenDone;

this->workItem = nullptr;

}

void SetJITCapReached()

{

Assert(this->GetState() == CodeGenQueued);

this->state = JITCapReached;

this->workItem = nullptr;

}

SmallSpanSequence\* GetNativeThrowSpanSequence() const

{

Assert(this->GetState() != NotScheduled);

Assert(this->GetState() != CleanedUp);

return nativeThrowSpanSequence;

}

void SetNativeThrowSpanSequence(SmallSpanSequence\* seq)

{

Assert(this->GetState() == CodeGenQueued);

Assert(this->nativeThrowSpanSequence == nullptr);

nativeThrowSpanSequence = seq;

}

bool IsInNativeAddressRange(DWORD\_PTR codeAddress) {

return (IsNativeCode() &&

codeAddress >= GetNativeAddress() &&

codeAddress < GetNativeAddress() + GetCodeSize());

}

#endif

DWORD\_PTR GetNativeAddress() const

{

// need the assert to skip for asmjsFunction as nativeAddress can be interpreter too for asmjs

Assert(this->GetState() == CodeGenRecorded || this->GetState() == CodeGenDone || this->isAsmJsFunction);

return (DWORD\_PTR)this->nativeAddress;

}

ptrdiff\_t GetCodeSize() const

{

Assert(this->GetState() == CodeGenRecorded || this->GetState() == CodeGenDone);

return codeSize;

}

CodeGenWorkItem \* GetWorkItem() const

{

State state = this->GetState();

Assert(state != NotScheduled || this->workItem == nullptr);

Assert(state == CleanedUp && this->workItem == nullptr ||

state != CleanedUp);

if (state == PendingCleanup)

{

return nullptr;

}

return this->workItem;

}

#ifndef TEMP\_DISABLE\_ASMJS

// set code size, used by TJ to set the code size

void SetCodeSize(ptrdiff\_t size)

{

Assert(isAsmJsFunction);

this->codeSize = size;

}

void SetNativeAddress(void\* address)

{

Assert(isAsmJsFunction);

this->nativeAddress = address;

}

void SetIsAsmJSFunction(bool value)

{

this->isAsmJsFunction = value;

}

#endif

bool GetIsAsmJSFunction()const

{

return this->isAsmJsFunction;

}

#ifndef TEMP\_DISABLE\_ASMJS

void SetTJCodeGenDone()

{

Assert(isAsmJsFunction);

this->state = CodeGenDone;

this->workItem = nullptr;

}

#endif

#if ENABLE\_NATIVE\_CODEGEN

void AddWeakFuncRef(RecyclerWeakReference<FunctionBody> \*weakFuncRef, Recycler \*recycler);

WeakFuncRefSet \*EnsureWeakFuncRefSet(Recycler \*recycler);

void EnsureIsReadyToCall();

void ProcessJitTransferData();

void ResetOnNativeCodeInstallFailure();

virtual void OnNativeCodeInstallFailure() = 0;

Js::PropertyGuard\* RegisterSharedPropertyGuard(Js::PropertyId propertyId, ScriptContext\* scriptContext);

bool HasSharedPropertyGuards() { return this->sharedPropertyGuards != nullptr; }

bool HasSharedPropertyGuard(Js::PropertyId propertyId);

bool TryGetSharedPropertyGuard(Js::PropertyId propertyId, Js::PropertyGuard\*& guard);

void RecordTypeGuards(int propertyGuardCount, TypeGuardTransferEntry\* typeGuardTransferRecord, size\_t typeGuardTransferPlusSize);

void RecordCtorCacheGuards(CtorCacheGuardTransferEntry\* ctorCacheTransferRecord, size\_t ctorCacheTransferPlusSize);

void FreePropertyGuards();

void FreeJitTransferData();

void RegisterEquivalentTypeCaches();

void UnregisterEquivalentTypeCaches();

bool ClearEquivalentTypeCaches();

void RegisterConstructorCache(Js::ConstructorCache\* constructorCache, Recycler\* recycler);

uint GetConstructorCacheCount() const { return this->constructorCaches != nullptr ? this->constructorCaches->Count() : 0; }

uint32 GetPendingPolymorphicCacheState() const { return this->pendingPolymorphicCacheState; }

void SetPendingPolymorphicCacheState(uint32 state) { this->pendingPolymorphicCacheState = state; }

BYTE GetPendingInlinerVersion() const { return this->pendingInlinerVersion; }

void SetPendingInlinerVersion(BYTE version) { this->pendingInlinerVersion = version; }

ImplicitCallFlags GetPendingImplicitCallFlags() const { return this->pendingImplicitCallFlags; }

void SetPendingImplicitCallFlags(ImplicitCallFlags flags) { this->pendingImplicitCallFlags = flags; }

virtual void Invalidate(bool prolongEntryPoint) { Assert(false); }

void RecordBailOutMap(JsUtil::List<LazyBailOutRecord, ArenaAllocator>\* bailoutMap);

void RecordInlineeFrameMap(JsUtil::List<NativeOffsetInlineeFramePair, ArenaAllocator>\* tempInlineeFrameMap);

InlineeFrameRecord\* FindInlineeFrame(void\* returnAddress);

bool HasInlinees() { return this->frameHeight > 0; }

void DoLazyBailout(BYTE\*\* addressOfeturnAddress, Js::FunctionBody\* functionBody, const PropertyRecord\* propertyRecord);

#endif

#if DBG\_DUMP

public:

#else if defined(VTUNE\_PROFILING)

private:

#endif

#if DBG\_DUMP | defined(VTUNE\_PROFILING)

// NativeOffsetMap is public for DBG\_DUMP, private for VTUNE\_PROFILING

struct NativeOffsetMap

{

uint32 statementIndex;

regex::Interval nativeOffsetSpan;

};

private:

JsUtil::List<NativeOffsetMap, HeapAllocator> nativeOffsetMaps;

public:

void RecordNativeMap(uint32 offset, uint32 statementIndex);

int GetNativeOffsetMapCount() const;

#endif

#if DBG\_DUMP && ENABLE\_NATIVE\_CODEGEN

void DumpNativeOffsetMaps();

void DumpNativeThrowSpanSequence();

NativeOffsetMap\* GetNativeOffsetMap(int index)

{

Assert(index >= 0);

Assert(index < GetNativeOffsetMapCount());

return &nativeOffsetMaps.Item(index);

}

#endif

#ifdef VTUNE\_PROFILING

public:

uint PopulateLineInfo(void\* pLineInfo, FunctionBody\* body);

#endif

protected:

void\* validationCookie;

};

class FunctionEntryPointInfo : public EntryPointInfo

{

public:

FunctionProxy \* functionProxy;

FunctionEntryPointInfo\* nextEntryPoint;

// The offset on the native stack, from which the locals are located (Populated at RegAlloc phase). Used for debug purpose.

int32 localVarSlotsOffset;

// The offset which stores that any of the locals are changed from the debugger.

int32 localVarChangedOffset;

uint entryPointIndex;

uint8 callsCount;

uint8 lastCallsCount;

bool nativeEntryPointProcessed;

private:

ExecutionMode jitMode;

FunctionEntryPointInfo\* mOldFunctionEntryPointInfo; // strong ref to oldEntryPointInfo(Int or TJ) in asm to ensure we don't collect it before JIT is completed

bool mIsTemplatizedJitMode; // true only if in TJ mode, used only for debugging

public:

static const uint8 GetDecrCallCountPerBailout()

{

return (100 / (uint8)CONFIG\_FLAG(RejitRatioLimit)) + 1;

}

FunctionEntryPointInfo(FunctionProxy \* functionInfo, void \* address, ThreadContext\* context, void\* validationCookie);

#ifndef TEMP\_DISABLE\_ASMJS

//AsmJS Support

void SetOldFunctionEntryPointInfo(FunctionEntryPointInfo\* entrypointInfo);

FunctionEntryPointInfo\* GetOldFunctionEntryPointInfo()const;

void SetIsTJMode(bool value);

bool GetIsTJMode()const;

//End AsmJS Support

#endif

virtual FunctionBody \*GetFunctionBody() const override;

#if ENABLE\_NATIVE\_CODEGEN

ExecutionMode GetJitMode() const;

void SetJitMode(const ExecutionMode jitMode);

virtual void Invalidate(bool prolongEntryPoint) override;

virtual void Expire() override;

virtual void EnterExpirableCollectMode() override;

virtual void OnNativeCodeInstallFailure() override;

#endif

virtual void OnCleanup(bool isShutdown) override;

virtual void ReleasePendingWorkItem() override;

#ifdef PERF\_COUNTERS

virtual void OnRecorded() override;

#endif

};

class LoopEntryPointInfo : public EntryPointInfo

{

public:

LoopHeader\* loopHeader;

LoopEntryPointInfo(LoopHeader\* loopHeader, Js::JavascriptLibrary\* library, void\* validationCookie) :

loopHeader(loopHeader), mIsTemplatizedJitMode(false),EntryPointInfo(nullptr, library, validationCookie, /\*threadContext\*/ nullptr, /\*isLoopBody\*/ true)

#ifdef BGJIT\_STATS

,used(false)

#endif

{ }

virtual FunctionBody \*GetFunctionBody() const override;

virtual void OnCleanup(bool isShutdown) override;

#if ENABLE\_NATIVE\_CODEGEN

virtual void OnNativeCodeInstallFailure() override;

#endif

#ifndef TEMP\_DISABLE\_ASMJS

void SetIsTJMode(bool value)

{

Assert(this->GetIsAsmJSFunction());

mIsTemplatizedJitMode = value;

}

bool GetIsTJMode()const

{

return mIsTemplatizedJitMode;

};

#endif

#ifdef PERF\_COUNTERS

virtual void OnRecorded() override;

#endif

#ifdef BGJIT\_STATS

bool IsUsed() const

{

return this->used;

}

void MarkAsUsed()

{

this->used = true;

}

#endif

private:

#ifdef BGJIT\_STATS

bool used;

#endif

bool mIsTemplatizedJitMode;

};

typedef RecyclerWeakReference<FunctionEntryPointInfo> FunctionEntryPointWeakRef;

typedef SynchronizableList<FunctionEntryPointWeakRef\*, JsUtil::List<FunctionEntryPointWeakRef\*>> FunctionEntryPointList;

typedef SynchronizableList<LoopEntryPointInfo\*, JsUtil::List<LoopEntryPointInfo\*>> LoopEntryPointList;

#pragma endregion

struct LoopHeader

{

private:

LoopEntryPointList\* entryPoints;

public:

uint startOffset;

uint endOffset;

uint interpretCount;

uint profiledLoopCounter;

bool isNested;

bool isInTry;

FunctionBody \* functionBody;

#if DBG\_DUMP

uint nativeCount;

#endif

static const uint NoLoop = (uint)-1;

static const uint GetOffsetOfProfiledLoopCounter() { return offsetof(LoopHeader, profiledLoopCounter); }

static const uint GetOffsetOfInterpretCount() { return offsetof(LoopHeader, interpretCount); }

bool Contains(Js::LoopHeader \* loopHeader) const

{

return (this->startOffset <= loopHeader->startOffset && loopHeader->endOffset <= this->endOffset);

}

bool Contains(uint offset) const

{

return this->startOffset <= offset && offset < this->endOffset;

}

void \* GetCurrentEntryPoint() const

{

LoopEntryPointInfo \* entryPoint = GetCurrentEntryPointInfo();

if (entryPoint != nullptr)

{

return this->entryPoints->Item(this->GetCurrentEntryPointIndex())->address;

}

return nullptr;

}

LoopEntryPointInfo \* GetCurrentEntryPointInfo() const

{

Assert(this->entryPoints->Count() > 0);

return this->entryPoints->Item(this->GetCurrentEntryPointIndex());

}

uint GetByteCodeCount()

{

return (endOffset - startOffset);

}

int GetCurrentEntryPointIndex() const

{

return this->entryPoints->Count() - 1;

}

LoopEntryPointInfo \* GetEntryPointInfo(int index) const

{

return this->entryPoints->Item(index);

}

template <class Fn>

void MapEntryPoints(Fn fn) const

{

if (this->entryPoints) // ETW rundown may call this before entryPoints initialization

{

this->entryPoints->Map([&](int index, LoopEntryPointInfo \* entryPoint)

{

if (entryPoint != nullptr)

{

fn(index, entryPoint);

}

});

}

}

template <class DebugSite, class Fn>

HRESULT MapEntryPoints(DebugSite site, Fn fn) const // external debugging version

{

return Map(site, this->entryPoints, [&](int index, LoopEntryPointInfo \* entryPoint)

{

if (entryPoint != nullptr)

{

fn(index, entryPoint);

}

});

}

void Init(FunctionBody \* functionBody);

#if ENABLE\_NATIVE\_CODEGEN

int CreateEntryPoint();

void ReleaseEntryPoints();

#endif

void ResetInterpreterCount()

{

this->interpretCount = 0;

}

void ResetProfiledLoopCounter()

{

this->profiledLoopCounter = 0;

}

};

class FunctionProxy;

#ifdef RECYCLER\_WRITE\_BARRIER

CompileAssert(sizeof(WriteBarrierPtr<FunctionProxy>) == sizeof(FunctionProxy\*));

typedef WriteBarrierPtr<FunctionProxy>\* FunctionProxyArray;

typedef WriteBarrierPtr<FunctionProxy>\* FunctionProxyPtrPtr;

#else

typedef FunctionProxy\*\* FunctionProxyArray;

typedef FunctionProxy\*\* FunctionProxyPtrPtr;

#endif

//

// FunctionProxy represents a user defined function

// This could be either from a source file or the byte code cache

// The function need not have been compiled yet- it could be parsed or compiled

// at a later time

//

class FunctionProxy : public FunctionInfo

{

protected:

FunctionProxy(JavascriptMethod entryPoint, Attributes attributes, int nestedCount, int derivedSize,

LocalFunctionId functionId, ScriptContext\* scriptContext, Utf8SourceInfo\* utf8SourceInfo, uint functionNumber);

DEFINE\_VTABLE\_CTOR\_NO\_REGISTER(FunctionProxy, FunctionInfo);

public:

enum SetDisplayNameFlags

{

SetDisplayNameFlagsNone = 0,

SetDisplayNameFlagsDontCopy = 1,

SetDisplayNameFlagsRecyclerAllocated = 2

};

typedef RecyclerWeakReference<DynamicType> FunctionTypeWeakRef;

typedef JsUtil::List<FunctionTypeWeakRef\*, Recycler, false, WeakRefFreeListedRemovePolicy> FunctionTypeWeakRefList;

uint32 GetSourceContextId() const;

wchar\_t\* GetDebugNumberSet(wchar(&bufferToWriteTo)[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE]) const;

bool GetIsTopLevel() { return m\_isTopLevel; }

void SetIsTopLevel(bool set) { m\_isTopLevel = set; }

bool GetIsAnonymousFunction() const { return this->GetDisplayName() == Js::Constants::AnonymousFunction; }

uint GetNestedCount() const { return m\_nestedCount; }

void Copy(FunctionProxy\* other);

ParseableFunctionInfo\* EnsureDeserialized();

ScriptContext\* GetScriptContext() const;

Utf8SourceInfo\* GetUtf8SourceInfo() const { return this->m\_utf8SourceInfo; }

void SetReferenceInParentFunction(FunctionProxyPtrPtr reference);

void UpdateReferenceInParentFunction(FunctionProxy\* newFunctionInfo);

DWORD\_PTR GetSecondaryHostSourceContext() const;

DWORD\_PTR GetHostSourceContext() const;

SourceContextInfo \* GetSourceContextInfo() const;

SRCINFO const \* GetHostSrcInfo() const;

uint GetFunctionNumber() const { return m\_functionNumber; }

virtual void Finalize(bool isShutdown) override;

void UpdateFunctionBodyImpl(FunctionBody\* body);

bool IsFunctionBody() const;

ProxyEntryPointInfo\* GetDefaultEntryPointInfo() const;

ScriptFunctionType \* GetDeferredPrototypeType() const;

ScriptFunctionType \* EnsureDeferredPrototypeType();

JavascriptMethod GetDirectEntryPoint(ProxyEntryPointInfo\* entryPoint) const;

// Function object type list methods

FunctionTypeWeakRefList\* EnsureFunctionObjectTypeList();

void RegisterFunctionObjectType(DynamicType\* functionType);

template <typename Fn>

void MapFunctionObjectTypes(Fn func);

static uint GetOffsetOfDeferredPrototypeType() { return offsetof(Js::FunctionProxy, deferredPrototypeType); }

static Js::ScriptFunctionType \* EnsureFunctionProxyDeferredPrototypeType(FunctionProxy \* proxy)

{

return proxy->EnsureDeferredPrototypeType();

}

void SetIsPublicLibraryCode() { m\_isPublicLibraryCode = true; }

bool IsPublicLibraryCode() const { return m\_isPublicLibraryCode; }

#if DBG

bool HasValidEntryPoint() const;

bool HasValidProfileEntryPoint() const;

bool HasValidNonProfileEntryPoint() const;

#endif

virtual void SetDisplayName(const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, SetDisplayNameFlags flags = SetDisplayNameFlagsNone) = 0;

virtual const wchar\_t\* GetDisplayName() const = 0;

virtual uint GetDisplayNameLength() const = 0;

virtual uint GetShortDisplayNameOffset() const = 0;

static const wchar\_t\* WrapWithBrackets(const wchar\_t\* name, charcount\_t sz, ScriptContext\* scriptContext);

// Used only in the library function stringify (toString, DiagGetValueString).

// If we need more often to give the short name, we should create a member variable which points to the short name

// this is also now being used for function.name.

const wchar\_t\* GetShortDisplayName(charcount\_t \* shortNameLength);

bool IsJitLoopBodyPhaseEnabled() const

{

// Consider: Allow JitLoopBody in generator functions for loops that do not yield.

return !PHASE\_OFF(JITLoopBodyPhase, this) && DoFullJit() && !this->IsGenerator();

}

bool IsJitLoopBodyPhaseForced() const

{

return

IsJitLoopBodyPhaseEnabled() &&

(

PHASE\_FORCE(JITLoopBodyPhase, this)

#ifdef ENABLE\_PREJIT

|| Configuration::Global.flags.Prejit

#endif

);

}

ULONG GetHostStartLine() const;

ULONG GetHostStartColumn() const;

bool DoFullJit() const

{

return !PHASE\_OFF(FullJitPhase, this);

}

protected:

// Static method(s)

static void SetDisplayName(const wchar\_t\* srcName, WriteBarrierPtr<const wchar\_t>\* destName, uint displayNameLength, ScriptContext \* scriptContext, SetDisplayNameFlags flags = SetDisplayNameFlagsNone);

static bool SetDisplayName(const wchar\_t\* srcName, const wchar\_t\*\* destName, uint displayNameLength, ScriptContext \* scriptContext, SetDisplayNameFlags flags = SetDisplayNameFlagsNone);

static bool IsConstantFunctionName(const wchar\_t\* srcName);

protected:

NoWriteBarrierPtr<ScriptContext> m\_scriptContext; // Memory context for this function body

WriteBarrierPtr<Utf8SourceInfo> m\_utf8SourceInfo;

// WriteBarrier-TODO: Consider changing this to NoWriteBarrierPtr, and skip tagging- also, tagging is likely unnecessary since that pointer in question is likely not resolvable

FunctionProxyPtrPtr m\_referenceInParentFunction; // Reference to nested function reference to this function in the parent function body (tagged to not be actual reference)

WriteBarrierPtr<ScriptFunctionType> deferredPrototypeType;

// Script function types not including the deferred prototype type

WriteBarrierPtr<FunctionTypeWeakRefList> m\_functionObjectTypeList;

WriteBarrierPtr<ProxyEntryPointInfo> m\_defaultEntryPointInfo; // The default entry point info for the function proxy

NoWriteBarrierField<uint> m\_derivedSize;

NoWriteBarrierField<uint> m\_functionNumber; // Per thread global function number

#define DEFINE\_FUNCTION\_PROXY\_FIELDS 1

#define CURRENT\_ACCESS\_MODIFIER protected:

#include "SerializableFunctionFields.h"

bool m\_isTopLevel : 1; // Indicates that this function is top-level function, currently being used in script profiler and debugger

bool m\_isPublicLibraryCode: 1; // Indicates this function is public boundary library code that should be visible in JS stack

void CleanupFunctionProxyCounters()

{

PERF\_COUNTER\_DEC(Code, TotalFunction);

}

ULONG ComputeAbsoluteLineNumber(ULONG relativeLineNumber) const;

ULONG ComputeAbsoluteColumnNumber(ULONG relativeLineNumber, ULONG relativeColumnNumber) const;

ULONG GetLineNumberInHostBuffer(ULONG relativeLineNumber) const;

private:

ScriptFunctionType \* AllocDeferredPrototypeType();

};

// Represents a function from the byte code cache which will

// be deserialized upon use

class DeferDeserializeFunctionInfo: public FunctionProxy

{

friend struct ByteCodeSerializer;

private:

DeferDeserializeFunctionInfo(int nestedFunctionCount, LocalFunctionId functionId, ByteCodeCache\* byteCodeCache, const byte\* serializedFunction, Utf8SourceInfo\* sourceInfo, ScriptContext\* scriptContext, uint functionNumber, const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, NativeModule \*nativeModule, Attributes attributes);

public:

static DeferDeserializeFunctionInfo\* New(ScriptContext\* scriptContext, int nestedFunctionCount, LocalFunctionId functionId, ByteCodeCache\* byteCodeCache, const byte\* serializedFunction, Utf8SourceInfo\* utf8SourceInfo, const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, NativeModule \*nativeModule, Attributes attributes);

virtual void Finalize(bool isShutdown) override;

FunctionBody\* Deserialize();

virtual const wchar\_t\* GetDisplayName() const override;

void SetDisplayName(const wchar\_t\* displayName);

virtual void SetDisplayName(const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, SetDisplayNameFlags flags = SetDisplayNameFlagsNone) override;

virtual uint GetDisplayNameLength() const { return m\_displayNameLength; }

virtual uint GetShortDisplayNameOffset() const { return m\_displayShortNameOffset; }

LPCWSTR GetSourceInfo(int& lineNumber, int& columnNumber) const;

private:

const byte\* m\_functionBytes;

ByteCodeCache\* m\_cache;

const wchar\_t \* m\_displayName; // Optional name

uint m\_displayNameLength;

uint m\_displayShortNameOffset;

NativeModule \*m\_nativeModule;

};

class ParseableFunctionInfo: public FunctionProxy

{

friend class ByteCodeBufferReader;

protected:

ParseableFunctionInfo(JavascriptMethod method, int nestedFunctionCount, int derivedSize, LocalFunctionId functionId, Utf8SourceInfo\* sourceInfo, ScriptContext\* scriptContext, uint functionNumber, const wchar\_t\* displayName, uint m\_displayNameLength, uint displayShortNameOffset, Attributes attributes, Js::PropertyRecordList\* propertyRecordList);

public:

static ParseableFunctionInfo\* New(ScriptContext\* scriptContext, int nestedFunctionCount, LocalFunctionId functionId, Utf8SourceInfo\* utf8SourceInfo, const wchar\_t\* displayName, uint m\_displayNameLength, uint displayShortNameOffset, Js::PropertyRecordList\* propertyRecordList, Attributes attributes);

DEFINE\_VTABLE\_CTOR\_NO\_REGISTER(ParseableFunctionInfo, FunctionProxy);

FunctionBody\* Parse(ScriptFunction \*\* functionRef = nullptr, bool isByteCodeDeserialization = false);

#ifndef TEMP\_DISABLE\_ASMJS

FunctionBody\* ParseAsmJs(Parser \* p, \_\_out CompileScriptException \* se, \_\_out ParseNodePtr \* ptree);

#endif

virtual uint GetDisplayNameLength() const { return m\_displayNameLength; }

virtual uint GetShortDisplayNameOffset() const { return m\_displayShortNameOffset; }

bool GetIsDeclaration() const { return m\_isDeclaration; }

void SetIsDeclaration(const bool is) { m\_isDeclaration = is; }

bool GetIsAccessor() const { return m\_isAccessor; }

void SetIsAccessor(const bool is) { m\_isAccessor = is; }

bool GetIsGlobalFunc() const { return m\_isGlobalFunc; }

void SetIsStaticNameFunction(const bool is) { m\_isStaticNameFunction = is; }

bool GetIsStaticNameFunction() const { return m\_isStaticNameFunction; }

void SetIsNamedFunctionExpression(const bool is) { m\_isNamedFunctionExpression = is; }

bool GetIsNamedFunctionExpression() const { return m\_isNamedFunctionExpression; }

void SetIsNameIdentifierRef (const bool is) { m\_isNameIdentifierRef = is; }

bool GetIsNameIdentifierRef () const { return m\_isNameIdentifierRef ; }

// Fake global ->

// 1) new Function code's global code

// 2) global code generated from the reparsing deferred parse function

bool IsFakeGlobalFunc(ulong flags) const;

void SetIsGlobalFunc(bool is) { m\_isGlobalFunc = is; }

bool GetIsStrictMode() const { return m\_isStrictMode; }

void SetIsStrictMode() { m\_isStrictMode = true; }

bool GetIsAsmjsMode() const { return m\_isAsmjsMode; }

void SetIsAsmjsMode(bool value)

{

m\_isAsmjsMode = value;

#if DBG

if (value)

{

m\_wasEverAsmjsMode = true;

}

#endif

}

bool GetHasImplicitArgIns() { return m\_hasImplicitArgIns; }

void SetHasImplicitArgIns(bool has) { m\_hasImplicitArgIns = has; }

ulong GetGrfscr() const;

void SetGrfscr(ulong grfscr);

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

///

/// ParseableFunctionInfo::GetInParamsCount

///

/// GetInParamsCount() returns the number of "in parameters" that have

/// currently been declared for this function:

/// - If this is "RegSlot\_VariableCount", the function takes a variable number

/// of parameters.

///

/// Consider: Change to store type information about parameters- names, type,

/// direction, etc.

///

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

ArgSlot GetInParamsCount() const { return m\_inParamCount; }

void SetInParamsCount(ArgSlot newInParamCount);

ArgSlot GetReportedInParamsCount() const;

void SetReportedInParamsCount(ArgSlot newReportedInParamCount);

void ResetInParams();

ScopeInfo\* GetScopeInfo() const { return this->m\_scopeInfo; }

void SetScopeInfo(ScopeInfo\* scopeInfo) { this->m\_scopeInfo = scopeInfo; }

PropertyId GetOrAddPropertyIdTracked(JsUtil::CharacterBuffer<WCHAR> const& propName);

bool IsTrackedPropertyId(PropertyId pid);

Js::PropertyRecordList\* GetBoundPropertyRecords() { return this->m\_boundPropertyRecords; }

void SetBoundPropertyRecords(Js::PropertyRecordList\* boundPropertyRecords)

{

Assert(this->m\_boundPropertyRecords == nullptr);

this->m\_boundPropertyRecords = boundPropertyRecords;

}

void ClearBoundPropertyRecords()

{

this->m\_boundPropertyRecords = nullptr;

}

ParseableFunctionInfo\* Clone(ScriptContext \*scriptContext, uint sourceIndex = Js::Constants::InvalidSourceIndex);

ParseableFunctionInfo\* CopyFunctionInfoInto(ScriptContext \*scriptContext, Js::ParseableFunctionInfo\* functionInfo, uint sourceIndex = Js::Constants::InvalidSourceIndex);

void CloneSourceInfo(ScriptContext\* scriptContext, const ParseableFunctionInfo& other, ScriptContext\* othersScriptContext, uint sourceIndex);

void SetInitialDefaultEntryPoint();

void SetDeferredParsingEntryPoint();

void SetEntryPoint(ProxyEntryPointInfo\* entryPoint, Js::JavascriptMethod address) {

entryPoint->address = address;

}

bool IsDynamicScript() const;

uint LengthInBytes() const { return m\_cbLength; }

uint StartOffset() const;

ULONG GetLineNumber() const;

ULONG GetColumnNumber() const;

template <class T>

LPCWSTR GetSourceName(const T& sourceContextInfo) const;

template <class T>

static LPCWSTR GetSourceName(const T& sourceContextInfo, bool m\_isEval, bool m\_isDynamicFunction);

LPCWSTR GetSourceName() const;

ULONG GetRelativeLineNumber() const { return m\_lineNumber; }

ULONG GetRelativeColumnNumber() const { return m\_columnNumber; }

uint GetSourceIndex() const;

LPCUTF8 GetSource(const wchar\_t\* reason = nullptr) const;

charcount\_t LengthInChars() const { return m\_cchLength; }

charcount\_t StartInDocument() const;

bool IsEval() const { return m\_isEval; }

bool IsDynamicFunction() const;

bool GetDontInline() { return m\_dontInline; }

void SetDontInline(bool is) { m\_dontInline = is; }

LPCUTF8 GetStartOfDocument(const wchar\_t\* reason = nullptr) const;

bool IsReparsed() const { return m\_reparsed; }

void SetReparsed(bool set) { m\_reparsed = set; }

bool IsByteCodeDebugMode() { return isByteCodeDebugMode; }

void SetIsByteCodeDebugMode(bool set) { isByteCodeDebugMode = set; }

bool GetExternalDisplaySourceName(BSTR\* sourceName);

void SetDoBackendArgumentsOptimization(bool set)

{

if (m\_doBackendArgumentsOptimization)

{

m\_doBackendArgumentsOptimization = set;

}

}

bool GetDoBackendArgumentsOptimization()

{

return m\_doBackendArgumentsOptimization;

}

bool IsFunctionParsed()

{

return !IsDeferredParseFunction() || m\_hasBeenParsed;

}

void SetFunctionParsed(bool hasBeenParsed)

{

m\_hasBeenParsed = hasBeenParsed;

}

void SetSourceInfo(uint sourceIndex, ParseNodePtr node, bool isEval, bool isDynamicFunction);

void Copy(FunctionBody\* other);

const wchar\_t\* GetExternalDisplayName() const;

//

// Algorithm to retrieve a function body's external display name. Template supports both

// local FunctionBody and ScriptDAC (debugging) scenarios.

//

template <class T>

static const wchar\_t\* GetExternalDisplayName(const T\* funcBody)

{

Assert(funcBody != nullptr);

Assert(funcBody->GetDisplayName() != nullptr);

return funcBody->GetDisplayName();

}

virtual const wchar\_t\* GetDisplayName() const override;

void SetDisplayName(const wchar\_t\* displayName);

virtual void SetDisplayName(const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, SetDisplayNameFlags flags = SetDisplayNameFlagsNone) override;

virtual void Finalize(bool isShutdown) override;

Var GetCachedSourceString()

{

return cachedSourceString;

}

void SetCachedSourceString(Var sourceString)

{

Assert(this->cachedSourceString == nullptr);

this->cachedSourceString = sourceString;

}

FunctionProxyArray GetNestedFuncArray();

FunctionProxy\* GetNestedFunc(uint index);

FunctionProxyPtrPtr GetNestedFuncReference(uint index);

ParseableFunctionInfo\* GetNestedFunctionForExecution(uint index);

void SetNestedFunc(FunctionProxy\* nestedFunc, uint index, ulong flags);

void ClearNestedFunctionParentFunctionReference();

void SetCapturesThis() { attributes = (Attributes)(attributes | Attributes::CapturesThis); }

bool GetCapturesThis() { return (attributes & Attributes::CapturesThis) != 0; }

void BuildDeferredStubs(ParseNode \*pnodeFnc);

DeferredFunctionStub \*GetDeferredStubs() const { return this->deferredStubs; }

void SetDeferredStubs(DeferredFunctionStub \*stub) { this->deferredStubs = stub; }

void RegisterFuncToDiag(ScriptContext \* scriptContext, wchar\_t const \* pszTitle);

protected:

static HRESULT MapDeferredReparseError(HRESULT& hrParse, const CompileScriptException& se);

bool m\_hasBeenParsed : 1; // Has function body been parsed- true for actual function bodies, false for deferparse

bool m\_isDeclaration : 1;

bool m\_isAccessor : 1; // Function is a property getter or setter

bool m\_isStaticNameFunction : 1;

bool m\_isNamedFunctionExpression : 1;

bool m\_isNameIdentifierRef : 1;

bool m\_isClassMember : 1;

bool m\_isStrictMode : 1;

bool m\_isAsmjsMode : 1;

bool m\_isAsmJsFunction : 1;

bool m\_isGlobalFunc : 1;

bool m\_doBackendArgumentsOptimization :1;

bool m\_isEval : 1; // Source code is in 'eval'

bool m\_isDynamicFunction : 1; // Source code is in 'Function'

bool m\_hasImplicitArgIns : 1;

bool m\_dontInline : 1; // Used by the JIT's inliner

// Indicates if the function has been reparsed for debug attach/detach scenario.

bool m\_reparsed : 1;

bool isByteCodeDebugMode : 1; // Whether last time generated bytecode was generated for debug mode.

// This field is not required for deferred parsing but because our thunks can't handle offsets > 128 bytes

// yet, leaving this here for now. We can look at optimizing the function info and function proxy structures some

// more and also fix our thunks to handle 8 bit offsets

NoWriteBarrierField<bool> m\_utf8SourceHasBeenSet; // start of UTF8-encoded source

NoWriteBarrierField<uint> m\_sourceIndex; // index into the scriptContext's list of saved sources

#if DYNAMIC\_INTERPRETER\_THUNK

void\* m\_dynamicInterpreterThunk; // Unique 'thunk' for every interpreted function - used for ETW symbol decoding.

#endif

NoWriteBarrierField<uint> m\_cbStartOffset; // pUtf8Source is this many bytes from the start of the scriptContext's source buffer.

// This is generally the same as m\_cchStartOffset unless the buffer has a BOM

#define DEFINE\_PARSEABLE\_FUNCTION\_INFO\_FIELDS 1

#define CURRENT\_ACCESS\_MODIFIER protected:

#include "SerializableFunctionFields.h"

ULONG m\_lineNumber;

ULONG m\_columnNumber;

WriteBarrierPtr<const wchar\_t> m\_displayName; // Optional name

uint m\_displayNameLength;

uint m\_displayShortNameOffset;

WriteBarrierPtr<ScopeInfo> m\_scopeInfo;

WriteBarrierPtr<PropertyRecordList> m\_boundPropertyRecords;

WriteBarrierVar cachedSourceString;

WriteBarrierPtr<DeferredFunctionStub> deferredStubs;

public:

#if DBG

bool m\_wasEverAsmjsMode; // has m\_isAsmjsMode ever been true

NoWriteBarrierField<Js::LocalFunctionId> deferredParseNextFunctionId;

#endif

#if DBG

NoWriteBarrierField<UINT> scopeObjectSize; // If the scope is an activation object - its size

#endif

};

//

// Algorithm to retrieve a function body's source name (url). Template supports both

// local FunctionBody and ScriptDAC (debugging) scenarios.

//

template <class T>

LPCWSTR ParseableFunctionInfo::GetSourceName(const T& sourceContextInfo) const

{

return GetSourceName<T>(sourceContextInfo, this->m\_isEval, this->m\_isDynamicFunction);

}

template <class T>

LPCWSTR ParseableFunctionInfo::GetSourceName(const T& sourceContextInfo, bool m\_isEval, bool m\_isDynamicFunction)

{

if (sourceContextInfo->IsDynamic())

{

if (m\_isEval)

{

return Constants::EvalCode;

}

else if (m\_isDynamicFunction)

{

return Constants::FunctionCode;

}

else

{

return Constants::UnknownScriptCode;

}

}

else

{

return sourceContextInfo->url;

}

}

class FunctionBody : public ParseableFunctionInfo

{

DEFINE\_VTABLE\_CTOR\_NO\_REGISTER(FunctionBody, ParseableFunctionInfo);

friend class ByteCodeBufferBuilder;

friend class ByteCodeBufferReader;

public:

// same as MachDouble, used in the Func.h

static const uint DIAGLOCALSLOTSIZE = 8;

struct StatementMap

{

StatementMap() : isSubexpression(false) {}

static StatementMap \* New(Recycler\* recycler)

{

return RecyclerNew(recycler, StatementMap);

}

regex::Interval sourceSpan;

regex::Interval byteCodeSpan;

bool isSubexpression;

};

// The type of StatementAdjustmentRecord.

// A bitmask that can be OR'ed of multiple values of the enum.

enum StatementAdjustmentType : ushort

{

SAT\_None = 0,

// Specifies an adjustment for next statement when going from current to next.

// Used for transitioning from current stmt to next during normal control-flow,

// such as offset of Br after if-block when there is else block present,

// when throw happens inside if and we ignore exceptions (next statement in the list

// would be 'else' but we need to pass flow control to Br target rather than entering 'else').

SAT\_FromCurrentToNext = 0x01,

// Specifies an adjustment for beginning of next statement.

// If there is adjustment record, the statement following it starts at specified offset and not at offset specified in statementMap.

// Used for set next statement from arbitrary location.

SAT\_NextStatementStart = 0x02,

SAT\_All = SAT\_FromCurrentToNext | SAT\_NextStatementStart

};

class StatementAdjustmentRecord

{

uint m\_byteCodeOffset;

StatementAdjustmentType m\_adjustmentType;

public:

StatementAdjustmentRecord();

StatementAdjustmentRecord(StatementAdjustmentType type, int byteCodeOffset);

StatementAdjustmentRecord(const StatementAdjustmentRecord& other);

uint GetByteCodeOffset();

StatementAdjustmentType GetAdjustmentType();

};

// Offset and entry/exit of a block that must be processed in new interpreter frame rather than current.

// Used for try and catch blocks.

class CrossFrameEntryExitRecord

{

uint m\_byteCodeOffset;

// true means enter, false means exit.

bool m\_isEnterBlock;

public:

CrossFrameEntryExitRecord();

CrossFrameEntryExitRecord(uint byteCodeOffset, bool isEnterBlock);

CrossFrameEntryExitRecord(const CrossFrameEntryExitRecord& other);

uint GetByteCodeOffset() const;

bool GetIsEnterBlock();

};

typedef JsUtil::List<Js::FunctionBody::StatementMap\*> StatementMapList;

// Note: isLeaf = true template param below means that recycler should not be used to dispose the items.

typedef JsUtil::List<StatementAdjustmentRecord, Recycler, /\* isLeaf = \*/ true> StatementAdjustmentRecordList;

typedef JsUtil::List<CrossFrameEntryExitRecord, Recycler, /\* isLeaf = \*/ true> CrossFrameEntryExitRecordList;

// Contains recorded at bytecode generation time information about statements and try-catch blocks.

// Used by debugger.

struct AuxStatementData

{

// Contains statement adjustment data:

// For given bytecode, following statement needs an adjustment, see StatementAdjustmentType for details.

StatementAdjustmentRecordList\* m\_statementAdjustmentRecords;

// Contain data about entry/exit of blocks that cause processing in different interpreter stack frame, such as try or catch.

CrossFrameEntryExitRecordList\* m\_crossFrameBlockEntryExisRecords;

AuxStatementData();

};

class SourceInfo

{

friend class RemoteFunctionBody;

friend class ByteCodeBufferReader;

friend class ByteCodeBufferBuilder;

public:

SmallSpanSequence \* pSpanSequence;

RegSlot frameDisplayRegister; // this register slot cannot be 0 so we use that sentinel value to indicate invalid

RegSlot objectRegister; // this register slot cannot be 0 so we use that sentinel value to indicate invalid

WriteBarrierPtr<ScopeObjectChain> pScopeObjectChain;

WriteBarrierPtr<ByteBlock> m\_probeBackingBlock; // NULL if no Probes, otherwise a copy of the unmodified the byte-codeblock //Delay

int32 m\_probeCount; // The number of installed probes (such as breakpoints).

// List of bytecode offset for the Branch bytecode.

WriteBarrierPtr<AuxStatementData> m\_auxStatementData;

SourceInfo():

frameDisplayRegister(0),

objectRegister(0),

pScopeObjectChain(nullptr),

m\_probeBackingBlock(nullptr),

m\_probeCount(0),

m\_auxStatementData(nullptr),

pSpanSequence(nullptr)

{

}

};

private:

WriteBarrierPtr<ByteBlock> auxBlock; // Optional auxiliary information

WriteBarrierPtr<ByteBlock> auxContextBlock; // Optional auxiliary context specific information

WriteBarrierPtr<ByteBlock> byteCodeBlock; // Function byte-code for script functions

WriteBarrierPtr<FunctionEntryPointList> entryPoints;

WriteBarrierPtr<Js::LoopHeader> loopHeaderArray;

WriteBarrierPtr<Var> m\_constTable;

WriteBarrierPtr<FunctionCodeGenRuntimeData\*> m\_codeGenRuntimeData;

WriteBarrierPtr<FunctionCodeGenRuntimeData\*> m\_codeGenGetSetRuntimeData;

WriteBarrierPtr<void\*> inlineCaches;

InlineCachePointerArray<PolymorphicInlineCache> polymorphicInlineCaches; // Contains the latest polymorphic inline caches

WriteBarrierPtr<PolymorphicInlineCache> polymorphicInlineCachesHead; // DList of all polymorphic inline caches that aren't finalized yet

WriteBarrierPtr<PropertyId> cacheIdToPropertyIdMap;

WriteBarrierPtr<PropertyId> referencedPropertyIdMap;

WriteBarrierPtr<UnifiedRegex::RegexPattern\*>literalRegexes;

WriteBarrierPtr<AsmJsFunctionInfo> asmJsFunctionInfo;

WriteBarrierPtr<AsmJsModuleInfo> asmJsModuleInfo;

// For SourceInfo

WriteBarrierPtr<PropertyId> propertyIdsForScopeSlotArray;

// This is used for showing locals for the current frame.

WriteBarrierPtr<PropertyIdOnRegSlotsContainer> propertyIdOnRegSlotsContainer;

#if DBG

#define InlineCacheTypeNone 0x00

#define InlineCacheTypeInlineCache 0x01

#define InlineCacheTypeIsInst 0x02

WriteBarrierPtr<byte> m\_inlineCacheTypes;

#endif

WriteBarrierPtr<StatementMapList> pStatementMaps;

public:

static DWORD GetConstTableOffset() { return offsetof(FunctionBody, m\_constTable); }

static DWORD GetAuxiliaryDataOffset() { return offsetof(FunctionBody, auxBlock); }

static DWORD GetAuxiliaryContextDataOffset() { return offsetof(FunctionBody, auxContextBlock); }

static DWORD GetObjLiteralTypesOffset() { return offsetof(FunctionBody, objLiteralTypes); }

static DWORD GetInlineCachesOffset() { return offsetof(FunctionBody, inlineCaches); }

static DWORD GetInlineCacheCountOffset() { return offsetof(FunctionBody, inlineCacheCount); }

static DWORD GetLiteralRegexesOffset() { return offsetof(FunctionBody, literalRegexes); }

static DWORD GetDerivedSizeOffset() { return offsetof(FunctionBody, m\_derivedSize); }

static DWORD GetReferencedPropertyIdMapOffset() { return offsetof(FunctionBody, referencedPropertyIdMap); }

static DWORD GetCacheIdToPropertyIdMapOffset() { return offsetof(FunctionBody, cacheIdToPropertyIdMap); }

static DWORD GetAsmJsTotalLoopCountOffset(){ return offsetof(FunctionBody, m\_asmJsTotalLoopCount); }

#if DBG

int m\_DEBUG\_executionCount; // Count of outstanding on InterpreterStackFrame

bool m\_nativeEntryPointIsInterpreterThunk; // NativeEntry entry point is in fact InterpreterThunk.

// Set by bgjit in OutOfMemory scenario during codegen.

#endif

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

NoWriteBarrierField<uint> regAllocStoreCount;

NoWriteBarrierField<uint> regAllocLoadCount;

NoWriteBarrierField<uint> callCountStats;

#endif

NoWriteBarrierField<uint> interpretedCount;

NoWriteBarrierField<uint> loopInterpreterLimit;

// >>>>>>WARNING! WARNING!<<<<<<<<<<

//

// If you add compile-time attributes to this set, be sure to add them to the attributes that are

// copied in FunctionBody::Clone

//

SourceInfo m\_sourceInfo; // position of the source

// Data needed by profiler:

NoWriteBarrierField<uint> m\_uScriptId; // Delay //Script Block it belongs to. This is function no. of the global function created by engine for each block

#if DBG

NoWriteBarrierField<int> m\_iProfileSession; // Script profile session the meta data of this function is reported to.

#endif // DEBUG

// R0 is reserved for the return value, R1 for the root object

static const RegSlot ReturnValueRegSlot = 0;

static const RegSlot RootObjectRegSlot = 1;

static const RegSlot FirstRegSlot = 1;

// This value be set on the stack (on a particular offset), when the frame value got changed.

static const int LocalsChangeDirtyValue = 1;

enum FunctionBodyFlags : byte

{

Flags\_None = 0x00,

Flags\_StackNestedFunc = 0x01,

Flags\_HasOrParentHasArguments = 0x02,

Flags\_HasTry = 0x04,

Flags\_HasThis = 0x08,

Flags\_NonUserCode = 0x10,

Flags\_HasOnlyThisStatements = 0x20,

Flags\_HasNoExplicitReturnValue = 0x40, // Returns undefined, i.e. has no return statements or return with no expression

Flags\_HasRestParameter = 0x80

};

#define DEFINE\_FUNCTION\_BODY\_FIELDS 1

#define CURRENT\_ACCESS\_MODIFIER public:

#include "SerializableFunctionFields.h"

private:

bool m\_tag : 1; // Used to tag the low bit to prevent possible GC false references

bool m\_nativeEntryPointUsed : 1; // Code might have been generated but not yet used.

bool hasDoneLoopBodyCodeGen : 1; // Code generated for loop body, but not necessary available to execute yet.

bool m\_isFuncRegistered : 1;

bool m\_isFuncRegisteredToDiag : 1; // Mentions the function's context is registered with diagprobe.

bool funcEscapes : 1;

bool m\_hasBailoutInstrInJittedCode : 1; // Indicates whether function has bailout instructions. Valid only if hasDoneCodeGen is true

bool m\_pendingLoopHeaderRelease : 1; // Indicates whether loop headers need to be released

bool hasExecutionDynamicProfileInfo : 1;

bool cleanedUp: 1;

bool sourceInfoCleanedUp: 1;

bool dontRethunkAfterBailout : 1;

bool disableInlineApply : 1;

bool disableInlineSpread : 1;

bool hasHotLoop: 1;

bool wasCalledFromLoop : 1;

bool hasNestedLoop : 1;

bool recentlyBailedOutOfJittedLoopBody : 1;

bool m\_firstFunctionObject: 1;

bool m\_inlineCachesOnFunctionObject: 1;

// Used for the debug re-parse. Saves state of function on the first parse, and restores it on a reparse. The state below is either dependent on

// the state of the script context, or on other factors like whether it was defer parsed or not.

bool m\_hasSetIsObject : 1;

// Used for the debug purpose, this info will be stored (in the non-debug mode), when a function has all locals marked as non-local-referenced.

// So when we got to no-refresh debug mode, and try to re-use the same function body we can then enforce all locals to be non-local-referenced.

bool m\_hasAllNonLocalReferenced : 1;

bool m\_hasFunExprNameReference : 1;

bool m\_ChildCallsEval : 1;

bool m\_CallsEval : 1;

bool m\_hasReferenceableBuiltInArguments : 1;

// Used in the debug purpose. This is to avoid setting all locals to non-local-referenced, multiple times for each child function.

bool m\_hasDoneAllNonLocalReferenced : 1;

// Used by the script profiler, once the function compiled is sent this will be set to true.

bool m\_hasFunctionCompiledSent : 1;

bool m\_isFromNativeCodeModule : 1;

bool m\_isPartialDeserializedFunction : 1;

bool m\_isAsmJsScheduledForFullJIT : 1;

#ifdef PERF\_COUNTERS

bool m\_isDeserializedFunction : 1;

#endif

#if DBG

// Indicates that nested functions can be allocated on the stack (but may not be)

bool m\_canDoStackNestedFunc : 1;

#endif

#if DBG

bool initializedExecutionModeAndLimits : 1;

#endif

#ifdef IR\_VIEWER

// whether IR Dump is enabled for this function (used by parseIR)

bool m\_isIRDumpEnabled : 1;

WriteBarrierPtr<Js::DynamicObject> m\_irDumpBaseObject;

#endif /\* IR\_VIEWER \*/

NoWriteBarrierField<uint8> bailOnMisingProfileCount;

NoWriteBarrierField<uint8> bailOnMisingProfileRejitCount;

NoWriteBarrierField<byte> inlineDepth; // Used by inlining to avoid recursively inlining functions excessively

NoWriteBarrierField<ExecutionMode> executionMode;

NoWriteBarrierField<uint16> interpreterLimit;

NoWriteBarrierField<uint16> autoProfilingInterpreter0Limit;

NoWriteBarrierField<uint16> profilingInterpreter0Limit;

NoWriteBarrierField<uint16> autoProfilingInterpreter1Limit;

NoWriteBarrierField<uint16> simpleJitLimit;

NoWriteBarrierField<uint16> profilingInterpreter1Limit;

NoWriteBarrierField<uint16> fullJitThreshold;

NoWriteBarrierField<uint16> fullJitRequeueThreshold;

NoWriteBarrierField<uint16> committedProfiledIterations;

NoWriteBarrierField<uint> m\_depth; // Indicates how many times the function has been entered (so increases by one on each recursive call, decreases by one when we're done)

WriteBarrierPtr<RecyclerWeakReference<FunctionBody>> stackNestedFuncParent;

// >>>>>>WARNING! WARNING!<<<<<<<<<<

//

// If you add compile-time attributes to the above set, be sure to add them to the attributes that are

// copied in FunctionBody::Clone

//

NoWriteBarrierPtr<Js::ByteCodeCache> byteCodeCache; // Not GC allocated so naked pointer

NoWriteBarrierField<int> serializationIndex;

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

static bool shareInlineCaches;

#endif

WriteBarrierPtr<DynamicType\*> objLiteralTypes;

WriteBarrierPtr<FunctionEntryPointInfo> defaultFunctionEntryPointInfo;

WriteBarrierPtr<FunctionEntryPointInfo> simpleJitEntryPointInfo;

#if ENABLE\_PROFILE\_INFO

WriteBarrierPtr<DynamicProfileInfo> dynamicProfileInfo;

WriteBarrierPtr<PolymorphicCallSiteInfo> polymorphicCallSiteInfoHead;

#endif

FunctionBailOutRecord \* functionBailOutRecord;

// select dynamic profile info saved off when we codegen and later

// used for rejit decisions (see bailout.cpp)

NoWriteBarrierField<BYTE> savedInlinerVersion;

#if ENABLE\_NATIVE\_CODEGEN

NoWriteBarrierField<ImplicitCallFlags> savedImplicitCallsFlags;

#endif

// State of inline caches (polymorphic vs. monomorphic) reflected in

// last jitted version of this func.

NoWriteBarrierField<uint32> savedPolymorphicCacheState;

// Used to track where we are when adding debugger scopes to the scope chain

// in order to avoid re-adding existing entries.

NoWriteBarrierField<int> debuggerScopeIndex;

FunctionBody(ScriptContext\* scriptContext, const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, uint nestedCount, Utf8SourceInfo\* sourceInfo,

uint uFunctionNumber, uint uScriptId, Js::LocalFunctionId functionId, Js::PropertyRecordList\* propRecordList, Attributes attributes

#ifdef PERF\_COUNTERS

, bool isDeserializedFunction = false

#endif

);

void SetNativeEntryPoint(FunctionEntryPointInfo\* entryPointInfo, JavascriptMethod originalEntryPoint, Var directEntryPoint);

#if DYNAMIC\_INTERPRETER\_THUNK

void GenerateDynamicInterpreterThunk();

#endif

void CloneByteCodeInto(ScriptContext \* scriptContext, FunctionBody \*newFunctionBody, uint sourceIndex);

void \* GetEntryPoint(ProxyEntryPointInfo\* entryPoint) const { return entryPoint->address; }

void CaptureDynamicProfileState(FunctionEntryPointInfo\* entryPointInfo);

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

void DumpRegStats(FunctionBody \*funcBody);

#endif

// Gets the next index for tracking debugger scopes (increments the internal counter as well).

int GetNextDebuggerScopeIndex();

public:

FunctionBody(ByteCodeCache\* cache, Utf8SourceInfo\* sourceInfo, ScriptContext\* scriptContext):

ParseableFunctionInfo((JavascriptMethod) nullptr, 0, 0, (LocalFunctionId) 0, sourceInfo, scriptContext, 0, nullptr, 0, 0, None, nullptr)

{

// Dummy constructor- does nothing

// Must be stack allocated

// Used during deferred bytecode serialization

}

static FunctionBody \* NewFromRecycler(Js::ScriptContext \* scriptContext, const wchar\_t \* displayName, uint displayNameLength, uint displayShortNameOffset, uint nestedCount,

Utf8SourceInfo\* sourceInfo, uint uScriptId, Js::LocalFunctionId functionId, Js::PropertyRecordList\* boundPropertyRecords, Attributes attributes

#ifdef PERF\_COUNTERS

, bool isDeserializedFunction

#endif

);

static FunctionBody \* NewFromRecycler(Js::ScriptContext \* scriptContext, const wchar\_t \* displayName, uint displayNameLength, uint displayShortNameOffset, uint nestedCount,

Utf8SourceInfo\* sourceInfo, uint uFunctionNumber, uint uScriptId, Js::LocalFunctionId functionId, Js::PropertyRecordList\* boundPropertyRecords, Attributes attributes

#ifdef PERF\_COUNTERS

, bool isDeserializedFunction

#endif

);

FunctionEntryPointInfo \* GetEntryPointInfo(int index) const;

FunctionEntryPointInfo \* TryGetEntryPointInfo(int index) const;

Js::RootObjectBase \* LoadRootObject() const;

Js::RootObjectBase \* GetRootObject() const;

ByteBlock\* GetAuxiliaryData();

ByteBlock\* GetAuxiliaryContextData();

ByteBlock\* GetByteCode();

ByteBlock\* GetOriginalByteCode(); // Returns original bytecode without probes (such as BPs).

const Js::ByteCodeCache \* GetByteCodeCache() const;

void SetSerializationIndex(int index) { Assert(serializationIndex == -1 && index != -1); serializationIndex = index; }

const int GetSerializationIndex() const;

uint GetByteCodeCount() const { return m\_byteCodeCount; }

uint GetByteCodeWithoutLDACount() const { return m\_byteCodeWithoutLDACount; }

uint GetByteCodeInLoopCount() const { return m\_byteCodeInLoopCount; }

uint16 GetEnvDepth() const { return m\_envDepth; }

void SetEnvDepth(uint16 depth) { m\_envDepth = depth; }

RegSlot GetEnvReg() const { return envRegister; }

void SetEnvReg(RegSlot reg) { Assert(envRegister == Constants::NoRegister); envRegister = this->MapRegSlot(reg); }

RegSlot GetThisRegForEventHandler() const { return thisRegisterForEventHandler; }

void SetThisRegForEventHandler(RegSlot reg) { Assert(thisRegisterForEventHandler == Constants::NoRegister); thisRegisterForEventHandler = this->MapRegSlot(reg); }

bool HasScopeObject() const { return hasScopeObject; }

void SetHasScopeObject(bool has) { hasScopeObject = has; }

void SetLocalClosureReg(RegSlot reg) { Assert(localClosureRegister == Constants::NoRegister); localClosureRegister = this->MapRegSlot(reg); }

RegSlot GetLocalClosureReg() const { return localClosureRegister; }

void SetLocalFrameDisplayReg(RegSlot reg) { Assert(localFrameDisplayRegister == Constants::NoRegister); localFrameDisplayRegister = this->MapRegSlot(reg); }

RegSlot GetLocalFrameDisplayReg() const { return localFrameDisplayRegister; /\*localClosureRegister == Constants::NoRegister ? Constants::NoRegister : localClosureRegister + 1;\*/ }

RegSlot FirstInnerScopeReg() const { Assert(firstInnerScopeRegister != Constants::NoRegister); return firstInnerScopeRegister; }

void SetFirstInnerScopeReg(RegSlot reg) { Assert(reg != Constants::NoRegister); firstInnerScopeRegister = this->MapRegSlot(reg); }

RegSlot GetFuncExprScopeReg() const { return funcExprScopeRegister; }

void SetFuncExprScopeReg(RegSlot reg) { Assert(reg != Constants::NoRegister); funcExprScopeRegister = this->MapRegSlot(reg); }

uint GetInnerScopeCount() const { return innerScopeCount; }

void SetInnerScopeCount(uint count) { innerScopeCount = count; }

bool HasCachedScopePropIds() const { return hasCachedScopePropIds; }

void SetHasCachedScopePropIds(bool has) { hasCachedScopePropIds = has; }

size\_t GetLoopBodyName(uint loopNumber, \_Out\_writes\_opt\_z\_(sizeInChars) WCHAR\* displayName, \_In\_ size\_t sizeInChars);

void AllocateLoopHeaders();

void ReleaseLoopHeaders();

Js::LoopHeader \* GetLoopHeader(uint index) const;

Js::Var GetLoopHeaderArrayPtr() const

{

Assert(this->loopHeaderArray != nullptr);

return this->loopHeaderArray;

}

#ifndef TEMP\_DISABLE\_ASMJS

void SetIsAsmJsFullJitScheduled(bool val){ m\_isAsmJsScheduledForFullJIT = val; }

bool GetIsAsmJsFullJitScheduled(){ return m\_isAsmJsScheduledForFullJIT; }

uint32 GetAsmJSTotalLoopCount() const

{

return m\_asmJsTotalLoopCount;

}

void SetIsAsmJsFunction(bool isAsmJsFunction)

{

m\_isAsmJsFunction = isAsmJsFunction;

}

#endif

const bool GetIsAsmJsFunction() const

{

return m\_isAsmJsFunction;

}

#ifndef TEMP\_DISABLE\_ASMJS

bool IsHotAsmJsLoop()

{

// Negative MinTemplatizedJitLoopRunCount treats all loops as hot asm loop

if (CONFIG\_FLAG(MinTemplatizedJitLoopRunCount) < 0 || m\_asmJsTotalLoopCount > static\_cast<uint>(CONFIG\_FLAG(MinTemplatizedJitLoopRunCount)))

{

return true;

}

return false;

}

#endif

private:

void ResetLoops();

public:

static bool Is(void\* ptr);

uint GetScriptId() const { return m\_uScriptId; }

void\* GetAddressOfScriptId() const

{

return (void\*)&m\_uScriptId;

}

uint8 \*GetCallsCountAddress(EntryPointInfo\* info) const

{

FunctionEntryPointInfo\* entryPoint = (FunctionEntryPointInfo\*) info;

return &entryPoint->callsCount;

}

FunctionEntryPointInfo\* GetDefaultFunctionEntryPointInfo() const;

void SetDefaultFunctionEntryPointInfo(FunctionEntryPointInfo\* entryPointInfo, const JavascriptMethod originalEntryPoint);

FunctionEntryPointInfo \*GetSimpleJitEntryPointInfo() const;

void SetSimpleJitEntryPointInfo(FunctionEntryPointInfo \*const entryPointInfo);

private:

void VerifyExecutionMode(const ExecutionMode executionMode) const;

public:

ExecutionMode GetDefaultInterpreterExecutionMode() const;

ExecutionMode GetExecutionMode() const;

ExecutionMode GetInterpreterExecutionMode(const bool isPostBailout);

void SetExecutionMode(const ExecutionMode executionMode);

private:

bool IsInterpreterExecutionMode() const;

public:

bool TryTransitionToNextExecutionMode();

void TryTransitionToNextInterpreterExecutionMode();

void SetIsSpeculativeJitCandidate();

bool TryTransitionToJitExecutionMode();

void TransitionToSimpleJitExecutionMode();

void TransitionToFullJitExecutionMode();

private:

void VerifyExecutionModeLimits();

void InitializeExecutionModeAndLimits();

public:

void ReinitializeExecutionModeAndLimits();

private:

void SetFullJitThreshold(const uint16 newFullJitThreshold, const bool skipSimpleJit = false);

void CommitExecutedIterations();

void CommitExecutedIterations(uint16 &limit, const uint executedIterations);

private:

uint16 GetSimpleJitExecutedIterations() const;

public:

void ResetSimpleJitLimitAndCallCount();

private:

void SetSimpleJitCallCount(const uint16 simpleJitLimit) const;

void ResetSimpleJitCallCount();

public:

uint16 GetProfiledIterations() const;

public:

void OnFullJitDequeued(const FunctionEntryPointInfo \*const entryPointInfo);

public:

void TraceExecutionMode(const char \*const eventDescription = nullptr) const;

void TraceInterpreterExecutionMode() const;

private:

void DoTraceExecutionMode(const char \*const eventDescription) const;

public:

static bool IsNewSimpleJit();

bool DoSimpleJit() const;

bool DoSimpleJitDynamicProfile() const;

private:

bool DoInterpreterProfile() const;

bool DoInterpreterAutoProfile() const;

public:

bool WasCalledFromLoop() const;

void SetWasCalledFromLoop();

public:

bool RecentlyBailedOutOfJittedLoopBody() const;

void SetRecentlyBailedOutOfJittedLoopBody(const bool value);

private:

static uint16 GetMinProfileIterations();

public:

static uint16 GetMinFunctionProfileIterations();

private:

static uint GetMinLoopProfileIterations(const uint loopInterpreterLimit);

public:

uint GetLoopProfileThreshold(const uint loopInterpreterLimit) const;

private:

static uint GetReducedLoopInterpretCount();

public:

uint GetLoopInterpretCount(LoopHeader\* loopHeader) const;

private:

static bool DoObjectHeaderInlining();

static bool DoObjectHeaderInliningForConstructors();

public:

static bool DoObjectHeaderInliningForConstructor(const uint32 inlineSlotCapacity);

private:

static bool DoObjectHeaderInliningForObjectLiterals();

public:

static bool DoObjectHeaderInliningForObjectLiteral(const uint32 inlineSlotCapacity);

static bool DoObjectHeaderInliningForObjectLiteral(const PropertyIdArray \*const propIds, ScriptContext \*const scriptContext);

static bool DoObjectHeaderInliningForEmptyObjects();

public:

#if DBG

int GetProfileSession() { return m\_iProfileSession; }

#endif

virtual void Finalize(bool isShutdown) override;

void Cleanup(bool isScriptContextClosing);

void CleanupSourceInfo(bool isScriptContextClosing);

template<bool IsScriptContextShutdown>

void CleanUpInlineCaches();

void CleanupRecyclerData(bool isRecyclerShutdown, bool doEntryPointCleanupCaptureStack);

#ifdef PERF\_COUNTERS

void CleanupPerfCounter();

#endif

virtual void Dispose(bool isShutdown) override { }

bool HasRejit() const

{

if(this->entryPoints)

{

return this->entryPoints->Count() > 1;

}

return false;

}

#pragma region SourceInfo Methods

void CopySourceInfo(ParseableFunctionInfo\* originalFunctionInfo);

void FinishSourceInfo();

RegSlot GetFrameDisplayRegister() const;

void SetFrameDisplayRegister(RegSlot frameDisplayRegister);

RegSlot GetObjectRegister() const;

void SetObjectRegister(RegSlot objectRegister);

bool HasObjectRegister() const { return GetObjectRegister() != 0; }

ScopeObjectChain \*GetScopeObjectChain() const;

void SetScopeObjectChain(ScopeObjectChain \*pScopeObjectChain);

// fetch the Catch scope object which encloses the passed bytecode offset, returns NULL otherwise

Js::DebuggerScope \* GetDiagCatchScopeObjectAt(int byteCodeOffset);

ByteBlock \*GetProbeBackingBlock();

void SetProbeBackingBlock(ByteBlock\* probeBackingBlock);

bool HasLineBreak() const;

bool HasLineBreak(charcount\_t start, charcount\_t end) const;

bool HasGeneratedFromByteCodeCache() const { return this->byteCodeCache != nullptr; }

bool EndsAfter(size\_t offset) const;

void TrackLoad(int ichMin);

SmallSpanSequence\* GetStatementMapSpanSequence() const { return m\_sourceInfo.pSpanSequence; }

void RecordStatementMap(StatementMap\* statementMap);

void RecordStatementMap(SmallSpanSequenceIter &iter, StatementData \* data);

void RecordLoad(int ichMin, int bytecodeAfterLoad);

DebuggerScope\* RecordStartScopeObject(DiagExtraScopesType scopeType, int start, RegSlot scopeLocation, int\* index = nullptr);

void RecordEndScopeObject(DebuggerScope\* currentScope, int end);

DebuggerScope\* AddScopeObject(DiagExtraScopesType scopeType, int start, RegSlot scopeLocation);

bool TryGetDebuggerScopeAt(int index, DebuggerScope\*& debuggerScope);

StatementMapList \* GetStatementMaps() const;

static StatementMap \* GetNextNonSubexpressionStatementMap(StatementMapList \*statementMapList, int & startingAtIndex);

static StatementMap \* GetPrevNonSubexpressionStatementMap(StatementMapList \*statementMapList, int & startingAtIndex);

void RecordStatementAdjustment(uint offset, StatementAdjustmentType adjType);

void RecordCrossFrameEntryExitRecord(uint byteCodeOffset, bool isEnterBlock);

// Find out an offset falls within the range. returns TRUE if found.

BOOL GetBranchOffsetWithin(uint start, uint end, StatementAdjustmentRecord\* record);

bool GetLineCharOffset(int byteCodeOffset, ULONG\* line, LONG\* charOffset, bool canAllocateLineCache = true);

bool GetLineCharOffsetFromStartChar(int startCharOfStatement, ULONG\* \_line, LONG\* \_charOffset, bool canAllocateLineCache = true);

// Given bytecode position, returns the start position of the statement and length of the statement.

bool GetStatementIndexAndLengthAt(int byteCodeOffset, UINT32\* statementIndex, UINT32\* statementLength);

// skip any utf-8/utf-16 byte-order-mark. Returns the number of chars skipped.

static charcount\_t SkipByteOrderMark(\_\_in\_bcount\_z(4) LPCUTF8& documentStart)

{

charcount\_t retValue = 0;

Assert(documentStart != nullptr);

if (documentStart[0] == 0xEF &&

documentStart[1] == 0xBB &&

documentStart[2] == 0xBF)

{

// UTF-8 - EF BB BF

// 3 bytes skipped - reports one char skipped

documentStart += 3;

retValue = 1;

}

else if ((documentStart[0] == 0xFF && documentStart[1] == 0xFE) ||

(documentStart[0] == 0xFE && documentStart[1] == 0xFF))

{

// UTF-16 LE - FF FE

// UTF-16 BE - FE FF

// 2 bytes skipped - reports one char skipped

documentStart += 2;

retValue = 1;

}

return retValue;

}

StatementMap\* GetMatchingStatementMapFromByteCode(int byteCodeOffset, bool ignoreSubexpressions = false);

int GetEnclosingStatementIndexFromByteCode(int byteCodeOffset, bool ignoreSubexpressions = false);

StatementMap\* GetEnclosingStatementMapFromByteCode(int byteCodeOffset, bool ignoreSubexpressions = false);

StatementMap\* GetMatchingStatementMapFromSource(int byteCodeOffset, int\* pMapIndex = nullptr);

void RecordFrameDisplayRegister(RegSlot slot);

void RecordObjectRegister(RegSlot slot);

CrossFrameEntryExitRecordList\* GetCrossFrameEntryExitRecords();

#ifdef VTUNE\_PROFILING

uint GetStartOffset(uint statementIndex) const;

ULONG GetSourceLineNumber(uint statementIndex);

#endif

#pragma endregion

// Field accessors

bool GetHasBailoutInstrInJittedCode() const { return this->m\_hasBailoutInstrInJittedCode; }

void SetHasBailoutInstrInJittedCode(bool hasBailout) { this->m\_hasBailoutInstrInJittedCode = hasBailout; }

bool GetCanReleaseLoopHeaders() const { return (this->m\_depth == 0); }

void SetPendingLoopHeaderRelease(bool pendingLoopHeaderRelease) { this->m\_pendingLoopHeaderRelease = pendingLoopHeaderRelease; }

bool GetIsFromNativeCodeModule() const { return m\_isFromNativeCodeModule; }

void SetIsFromNativeCodeModule(bool isFromNativeCodeModule) { m\_isFromNativeCodeModule = isFromNativeCodeModule; }

uint GetLoopNumber(LoopHeader const \* loopHeader) const;

bool GetHasAllocatedLoopHeaders() { return this->loopHeaderArray != nullptr; }

#if ENABLE\_NATIVE\_CODEGEN

Js::JavascriptMethod GetLoopBodyEntryPoint(Js::LoopHeader \* loopHeader, int entryPointIndex);

void SetLoopBodyEntryPoint(Js::LoopHeader \* loopHeader, EntryPointInfo\* entryPointInfo, Js::JavascriptMethod entryPoint);

#endif

void RestoreOldDefaultEntryPoint(FunctionEntryPointInfo\* oldEntryPoint, JavascriptMethod oldOriginalEntryPoint, FunctionEntryPointInfo\* newEntryPoint);

FunctionEntryPointInfo\* CreateNewDefaultEntryPoint();

void AddEntryPointToEntryPointList(FunctionEntryPointInfo\* entryPoint);

// Kind of entry point for original entry point

BOOL IsInterpreterThunk() const;

BOOL IsDynamicInterpreterThunk() const;

BOOL IsNativeOriginalEntryPoint() const;

bool IsSimpleJitOriginalEntryPoint() const;

#if DYNAMIC\_INTERPRETER\_THUNK

static BYTE GetOffsetOfDynamicInterpreterThunk() { return offsetof(FunctionBody, m\_dynamicInterpreterThunk); }

void\* GetDynamicInterpreterEntryPoint() const

{

return m\_dynamicInterpreterThunk;

}

bool HasInterpreterThunkGenerated() const

{

return m\_dynamicInterpreterThunk != nullptr;

}

DWORD GetDynamicInterpreterThunkSize() const;

#endif

bool GetHasHotLoop() const { return hasHotLoop; };

void SetHasHotLoop();

bool GetHasNestedLoop() const { return hasNestedLoop; };

void SetHasNestedLoop(bool nest) { hasNestedLoop = nest; };

bool IsInlineApplyDisabled();

void InitDisableInlineApply();

void SetDisableInlineApply(bool set);

bool IsInlineSpreadDisabled() const { return disableInlineSpread; }

void InitDisableInlineSpread() { disableInlineSpread = this->functionId != Js::Constants::NoFunctionId && PHASE\_OFF(Js::InlinePhase, this); }

void SetDisableInlineSpread(bool set) { disableInlineSpread = set; }

bool CheckCalleeContextForInlining(FunctionProxy\* calleeFunctionProxy);

#if DBG

bool HasValidSourceInfo();

#endif

#if DYNAMIC\_INTERPRETER\_THUNK

JavascriptMethod EnsureDynamicInterpreterThunk(FunctionEntryPointInfo\* entryPointInfo);

#endif

void SetCheckCodeGenEntryPoint(FunctionEntryPointInfo\* entryPointInfo, JavascriptMethod entryPoint);

#if ENABLE\_NATIVE\_CODEGEN

typedef void (\*SetNativeEntryPointFuncType)(FunctionEntryPointInfo\* entryPointInfo, Js::FunctionBody \* functionBody, Js::JavascriptMethod entryPoint);

static void DefaultSetNativeEntryPoint(FunctionEntryPointInfo\* entryPointInfo, FunctionBody \* functionBody, JavascriptMethod entryPoint);

static void ProfileSetNativeEntryPoint(FunctionEntryPointInfo\* entryPointInfo, FunctionBody \* functionBody, JavascriptMethod entryPoint);

bool GetNativeEntryPointUsed() const { return m\_nativeEntryPointUsed; }

void SetNativeEntryPointUsed(bool nativeEntryPointUsed) { this->m\_nativeEntryPointUsed = nativeEntryPointUsed; }

#endif

bool GetIsFuncRegistered() { return m\_isFuncRegistered; }

void SetIsFuncRegistered(bool isRegistered) { m\_isFuncRegistered = isRegistered; }

bool GetHasLoops() const { return loopCount != 0; }

uint IncrLoopCount() { return this->loopCount++; }

uint GetLoopCount() const { return this->loopCount; }

bool AllocProfiledDivOrRem(ProfileId\* profileId) { if (this->profiledDivOrRemCount != Constants::NoProfileId) { \*profileId = this->profiledDivOrRemCount++; return true; } return false; }

ProfileId GetProfiledDivOrRemCount() { return this->profiledDivOrRemCount; }

bool AllocProfiledSwitch(ProfileId\* profileId) { if (this->profiledSwitchCount != Constants::NoProfileId) { \*profileId = this->profiledSwitchCount++; return true; } return false; }

ProfileId GetProfiledSwitchCount() { return this->profiledSwitchCount; }

bool AllocProfiledCallSiteId(ProfileId\* profileId) { if (this->profiledCallSiteCount != Constants::NoProfileId) { \*profileId = this->profiledCallSiteCount++; return true; } return false; }

ProfileId GetProfiledCallSiteCount() const { return this->profiledCallSiteCount; }

void SetProfiledCallSiteCount(ProfileId callSiteId) { this->profiledCallSiteCount = callSiteId; }

bool AllocProfiledArrayCallSiteId(ProfileId\* profileId) { if (this->profiledArrayCallSiteCount != Constants::NoProfileId) { \*profileId = this->profiledArrayCallSiteCount++; return true; } return false; }

ProfileId GetProfiledArrayCallSiteCount() const { return this->profiledArrayCallSiteCount; }

bool AllocProfiledReturnTypeId(ProfileId\* profileId) { if (this->profiledReturnTypeCount != Constants::NoProfileId) { \*profileId = this->profiledReturnTypeCount++; return true; } return false; }

ProfileId GetProfiledReturnTypeCount() const { return this->profiledReturnTypeCount; }

bool AllocProfiledSlotId(ProfileId\* profileId) { if (this->profiledSlotCount != Constants::NoProfileId) { \*profileId = this->profiledSlotCount++; return true; } return false; }

ProfileId GetProfiledSlotCount() const { return this->profiledSlotCount; }

ProfileId AllocProfiledLdElemId(ProfileId\* profileId) { if (this->profiledLdElemCount != Constants::NoProfileId) { \*profileId = this->profiledLdElemCount++; return true; } return false; }

ProfileId GetProfiledLdElemCount() const { return this->profiledLdElemCount; }

bool AllocProfiledStElemId(ProfileId\* profileId) { if (this->profiledStElemCount != Constants::NoProfileId) { \*profileId = this->profiledStElemCount++; return true; } return false; }

ProfileId GetProfiledStElemCount() const { return this->profiledStElemCount; }

uint GetProfiledFldCount() const { return this->GetInlineCacheCount(); }

ArgSlot GetProfiledInParamsCount() const { return this->GetInParamsCount() > 1? this->GetInParamsCount() - 1 : 0; }

bool IsPartialDeserializedFunction() { return this->m\_isPartialDeserializedFunction; }

#ifdef PERF\_COUNTERS

bool IsDeserializedFunction() { return this->m\_isDeserializedFunction; }

#endif

#ifdef IR\_VIEWER

bool IsIRDumpEnabled() const { return this->m\_isIRDumpEnabled; }

void SetIRDumpEnabled(bool enabled) { this->m\_isIRDumpEnabled = enabled; }

Js::DynamicObject \* GetIRDumpBaseObject();

#endif /\* IR\_VIEWER \*/

#if ENABLE\_NATIVE\_CODEGEN

void SetPolymorphicCallSiteInfoHead(PolymorphicCallSiteInfo \*polyCallSiteInfo) { this->polymorphicCallSiteInfoHead = polyCallSiteInfo;}

PolymorphicCallSiteInfo \* GetPolymorphicCallSiteInfoHead() { return this->polymorphicCallSiteInfoHead;}

#endif

PolymorphicInlineCache \* GetPolymorphicInlineCachesHead() { return this->polymorphicInlineCachesHead; }

void SetPolymorphicInlineCachesHead(PolymorphicInlineCache \* cache) { this->polymorphicInlineCachesHead = cache; }

bool PolyInliningUsingFixedMethodsAllowedByConfigFlags(FunctionBody\* topFunctionBody)

{

return !PHASE\_OFF(Js::InlinePhase, this) && !PHASE\_OFF(Js::InlinePhase, topFunctionBody) &&

!PHASE\_OFF(Js::PolymorphicInlinePhase, this) && !PHASE\_OFF(Js::PolymorphicInlinePhase, topFunctionBody) &&

!PHASE\_OFF(Js::FixedMethodsPhase, this) && !PHASE\_OFF(Js::FixedMethodsPhase, topFunctionBody) &&

!PHASE\_OFF(Js::PolymorphicInlineFixedMethodsPhase, this) && !PHASE\_OFF(Js::PolymorphicInlineFixedMethodsPhase, topFunctionBody);

}

Js::PropertyId \* GetPropertyIdsForScopeSlotArray() const

{

return this->propertyIdsForScopeSlotArray;

}

void SetPropertyIdsForScopeSlotArray(Js::PropertyId \* propertyIdsForScopeSlotArray, uint scopeSlotCount)

{

this->scopeSlotArraySize = scopeSlotCount;

this->propertyIdsForScopeSlotArray = propertyIdsForScopeSlotArray;

}

Js::PropertyIdOnRegSlotsContainer \* GetPropertyIdOnRegSlotsContainer() const

{

return this->propertyIdOnRegSlotsContainer;

}

private:

void ResetProfileIds();

void SetFlags(bool does, FunctionBodyFlags newFlags)

{

if (does)

{

flags = (FunctionBodyFlags)(flags | newFlags);

}

else

{

flags = (FunctionBodyFlags)(flags & ~newFlags);

}

}

public:

bool GetHasThis() const { return (flags & Flags\_HasThis) != 0; }

void SetHasThis(bool has) { SetFlags(has, Flags\_HasThis); }

bool GetHasTry() const { return (flags & Flags\_HasTry) != 0; }

void SetHasTry(bool has) { SetFlags(has, Flags\_HasTry); }

bool GetHasFinally() const { return m\_hasFinally; }

void SetHasFinally(bool has){ m\_hasFinally = has; }

bool GetFuncEscapes() const { return funcEscapes; }

void SetFuncEscapes(bool does) { funcEscapes = does; }

bool GetHasOrParentHasArguments() const { return (flags & Flags\_HasOrParentHasArguments) != 0; }

void SetHasOrParentHasArguments(bool has) { SetFlags(has, Flags\_HasOrParentHasArguments); }

bool DoStackNestedFunc() const { return (flags & Flags\_StackNestedFunc) != 0; }

void SetStackNestedFunc(bool does) { SetFlags(does, Flags\_StackNestedFunc); }

#if DBG

bool CanDoStackNestedFunc() const { return m\_canDoStackNestedFunc; }

void SetCanDoStackNestedFunc() { m\_canDoStackNestedFunc = true; }

#endif

FunctionBody \* GetStackNestedFuncParent();

FunctionBody \* GetAndClearStackNestedFuncParent();

void ClearStackNestedFuncParent();

void SetStackNestedFuncParent(FunctionBody \* parentFunctionBody);

#if defined(\_M\_IX86) || defined(\_M\_X64)

bool DoStackClosure() const

{

return DoStackNestedFunc() && GetNestedCount() != 0 && !PHASE\_OFF(StackClosurePhase, this) && scopeSlotArraySize != 0 && GetEnvDepth() != (uint16)-1;

}

#else

bool DoStackClosure() const

{

return false;

}

#endif

bool DoStackFrameDisplay() const { return DoStackClosure(); }

bool DoStackScopeSlots() const { return DoStackClosure(); }

bool IsNonUserCode() const { return (flags & Flags\_NonUserCode) != 0; }

void SetIsNonUserCode(bool set);

bool GetHasNoExplicitReturnValue() { return (flags & Flags\_HasNoExplicitReturnValue) != 0; }

void SetHasNoExplicitReturnValue(bool has) { SetFlags(has, Flags\_HasNoExplicitReturnValue); }

bool GetHasOnlyThisStmts() const { return (flags & Flags\_HasOnlyThisStatements) != 0; }

void SetHasOnlyThisStmts(bool has) { SetFlags(has, Flags\_HasOnlyThisStatements); }

bool GetIsFirstFunctionObject() const { return m\_firstFunctionObject; }

void SetIsNotFirstFunctionObject() { m\_firstFunctionObject = false; }

bool GetInlineCachesOnFunctionObject() { return m\_inlineCachesOnFunctionObject; }

void SetInlineCachesOnFunctionObject(bool has) { m\_inlineCachesOnFunctionObject = has; }

bool GetHasRestParameter() const { return (flags & Flags\_HasRestParameter) != 0; }

void SetHasRestParameter() { SetFlags(true, Flags\_HasRestParameter); }

uint GetNumberOfRecursiveCallSites();

bool CanInlineRecursively(uint depth, bool tryAggressive = true);

public:

bool CanInlineAgain() const

{

// Block excessive recursive inlining of the same function

return inlineDepth < static\_cast<byte>(max(1, min(0xff, CONFIG\_FLAG(MaxFuncInlineDepth))));

}

void OnBeginInlineInto()

{

++inlineDepth;

}

void OnEndInlineInto()

{

--inlineDepth;

}

uint8 IncrementBailOnMisingProfileCount() { return ++bailOnMisingProfileCount; }

void ResetBailOnMisingProfileCount() { bailOnMisingProfileCount = 0; }

uint8 IncrementBailOnMisingProfileRejitCount() { return ++bailOnMisingProfileRejitCount; }

uint32 GetFrameHeight(EntryPointInfo\* entryPointInfo) const;

void SetFrameHeight(EntryPointInfo\* entryPointInfo, uint32 frameHeight);

RegSlot GetLocalsCount();

RegSlot GetConstantCount() const { return m\_constCount; }

void SetConstantCount(RegSlot cNewConstants);

RegSlot GetVarCount();

void SetVarCount(RegSlot cNewVars);

RegSlot MapRegSlot(RegSlot reg)

{

if (this->RegIsConst(reg))

{

reg = CONSTREG\_TO\_REGSLOT(reg);

Assert(reg < this->GetConstantCount());

}

else

{

reg += this->GetConstantCount();

}

return reg;

}

bool RegIsConst(RegSlot reg) { return reg > REGSLOT\_TO\_CONSTREG(this->GetConstantCount()); }

uint32 GetNonTempLocalVarCount();

uint32 GetFirstNonTempLocalIndex();

uint32 GetEndNonTempLocalIndex();

bool IsNonTempLocalVar(uint32 varIndex);

bool GetSlotOffset(RegSlot slotId, int32 \* slotOffset, bool allowTemp = false);

RegSlot GetOutParamsDepth();

void SetOutParamDepth(RegSlot cOutParamsDepth);

RegSlot GetYieldRegister();

RegSlot GetFirstTmpReg();

void SetFirstTmpReg(RegSlot firstTmpReg);

RegSlot GetTempCount();

Js::ModuleID GetModuleID() const;

void CreateConstantTable();

void RecordNullObject(RegSlot location);

void RecordUndefinedObject(RegSlot location);

void RecordTrueObject(RegSlot location);

void RecordFalseObject(RegSlot location);

void RecordIntConstant(RegSlot location, unsigned int val);

void RecordStrConstant(RegSlot location, LPCOLESTR psz, ulong cch);

void RecordFloatConstant(RegSlot location, double d);

void RecordNullDisplayConstant(RegSlot location);

void RecordStrictNullDisplayConstant(RegSlot location);

void InitConstantSlots(Var \*dstSlots);

Var GetConstantVar(RegSlot location);

void\* GetConstTable() const{return m\_constTable;}

void CloneConstantTable(FunctionBody \*newFunc);

void MarkScript(ByteBlock \* pblkByteCode, ByteBlock \* pblkAuxiliaryData, ByteBlock\* auxContextBlock,

uint byteCodeCount, uint byteCodeInLoopCount, uint byteCodeWithoutLDACount);

void BeginExecution();

void EndExecution();

SourceInfo \* GetSourceInfo() { return &this->m\_sourceInfo; }

bool InstallProbe(int offset);

bool UninstallProbe(int offset);

bool ProbeAtOffset(int offsest, OpCode\* pOriginalOpcode);

FunctionBody \* Clone(ScriptContext \*scriptContext, uint sourceIndex = Js::Constants::InvalidSourceIndex);

static bool ShouldShareInlineCaches() { return CONFIG\_FLAG(ShareInlineCaches); }

uint GetInlineCacheCount() const { return inlineCacheCount; }

uint GetRootObjectLoadInlineCacheStart() const { return rootObjectLoadInlineCacheStart; }

uint GetRootObjectLoadMethodInlineCacheStart() const { return rootObjectLoadMethodInlineCacheStart; }

uint GetRootObjectStoreInlineCacheStart() const { return rootObjectStoreInlineCacheStart; }

uint GetIsInstInlineCacheCount() const { return isInstInlineCacheCount; }

uint GetReferencedPropertyIdCount() const { return referencedPropertyIdCount; }

void AllocateInlineCache();

InlineCache \* GetInlineCache(uint index);

bool CanFunctionObjectHaveInlineCaches();

void\*\* GetInlineCaches();

#if DBG

byte\* GetInlineCacheTypes();

#endif

InlineCache \* GetRootObjectInlineCache(uint index);

IsInstInlineCache \* GetIsInstInlineCache(uint index);

PolymorphicInlineCache \* GetPolymorphicInlineCache(uint index);

PolymorphicInlineCache \* CreateNewPolymorphicInlineCache(uint index, PropertyId propertyId, InlineCache \* inlineCache);

PolymorphicInlineCache \* CreateBiggerPolymorphicInlineCache(uint index, PropertyId propertyId);

private:

void ResetInlineCaches();

PolymorphicInlineCache \* CreatePolymorphicInlineCache(uint index, uint16 size);

uint32 m\_asmJsTotalLoopCount;

public:

void CreateCacheIdToPropertyIdMap();

void CreateCacheIdToPropertyIdMap(uint rootObjectLoadInlineCacheStart, uint rootObjectLoadMethodInlineCacheStart, uint rootObjectStoreInlineCacheStart,

uint totalFieldAccessInlineCacheCount, uint isInstInlineCacheCount);

void SetPropertyIdForCacheId(uint cacheId, PropertyId propertyId);

PropertyId GetPropertyIdFromCacheId(uint cacheId)

{

Assert(this->cacheIdToPropertyIdMap);

Assert(cacheId < this->GetInlineCacheCount());

return this->cacheIdToPropertyIdMap[cacheId];

}

#if DBG

void VerifyCacheIdToPropertyIdMap();

#endif

void CreateReferencedPropertyIdMap(uint referencedPropertyIdCount);

void CreateReferencedPropertyIdMap();

PropertyId GetReferencedPropertyIdWithMapIndex(uint mapIndex);

void SetReferencedPropertyIdWithMapIndex(uint mapIndex, PropertyId propertyId);

PropertyId GetReferencedPropertyId(uint index);

#if DBG

void VerifyReferencedPropertyIdMap();

#endif

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

void DumpFullFunctionName();

void DumpFunctionId(bool pad);

uint GetTraceFunctionNumber() const;

#endif

public:

uint NewObjectLiteral();

void AllocateObjectLiteralTypeArray();

DynamicType \*\* GetObjectLiteralTypeRef(uint index);

uint NewLiteralRegex();

uint GetLiteralRegexCount() const;

void AllocateLiteralRegexArray();

UnifiedRegex::RegexPattern \*GetLiteralRegex(const uint index);

#ifndef TEMP\_DISABLE\_ASMJS

AsmJsFunctionInfo\* GetAsmJsFunctionInfo()const {return asmJsFunctionInfo; }

AsmJsFunctionInfo\* AllocateAsmJsFunctionInfo();

AsmJsModuleInfo\* GetAsmJsModuleInfo()const{ return asmJsModuleInfo; }

void ResetAsmJsInfo()

{

asmJsFunctionInfo = nullptr;

asmJsModuleInfo = nullptr;

}

bool IsAsmJSModule()const{ if (asmJsModuleInfo) return true; return false; }

AsmJsModuleInfo\* AllocateAsmJsModuleInfo();

#endif

void SetLiteralRegex(const uint index, UnifiedRegex::RegexPattern \*const pattern);

private:

void ResetLiteralRegexes();

void ResetObjectLiteralTypes();

public:

void ResetByteCodeGenState();

void ResetByteCodeGenVisitState();

void FindClosestStatements(long characterOffset, StatementLocation \*firstStatementLocation, StatementLocation \*secondStatementLocation);

#if ENABLE\_NATIVE\_CODEGEN

const FunctionCodeGenRuntimeData \*GetInlineeCodeGenRuntimeData(const ProfileId profiledCallSiteId) const;

const FunctionCodeGenRuntimeData \*GetInlineeCodeGenRuntimeDataForTargetInlinee(const ProfileId profiledCallSiteId, FunctionBody \*inlineeFuncBody) const;

FunctionCodeGenRuntimeData \*EnsureInlineeCodeGenRuntimeData(

Recycler \*const recycler,

\_\_in\_range(0, profiledCallSiteCount - 1) const ProfileId profiledCallSiteId,

FunctionBody \*const inlinee);

const FunctionCodeGenRuntimeData \*GetLdFldInlineeCodeGenRuntimeData(const InlineCacheIndex inlineCacheIndex) const;

FunctionCodeGenRuntimeData \*EnsureLdFldInlineeCodeGenRuntimeData(

Recycler \*const recycler,

\_\_in\_range(0, this->inlineCacheCount - 1) const InlineCacheIndex inlineCacheIndex,

FunctionBody \*const inlinee);

void LoadDynamicProfileInfo();

bool HasExecutionDynamicProfileInfo() const { return hasExecutionDynamicProfileInfo; }

bool HasDynamicProfileInfo() const { return dynamicProfileInfo != nullptr; }

bool NeedEnsureDynamicProfileInfo() const;

DynamicProfileInfo \* GetDynamicProfileInfo() const { Assert(HasExecutionDynamicProfileInfo()); return dynamicProfileInfo; }

DynamicProfileInfo \* GetAnyDynamicProfileInfo() const { Assert(HasDynamicProfileInfo()); return dynamicProfileInfo; }

DynamicProfileInfo \* EnsureDynamicProfileInfo();

DynamicProfileInfo \* AllocateDynamicProfile();

BYTE GetSavedInlinerVersion() const;

uint32 GetSavedPolymorphicCacheState() const;

ImplicitCallFlags GetSavedImplicitCallsFlags() const;

bool HasNonBuiltInCallee();

bool HasFunctionBailOutRecord() const { return functionBailOutRecord != nullptr; }

FunctionBailOutRecord \* GetFunctionBailOutRecord() const { Assert(HasFunctionBailOutRecord()); return functionBailOutRecord; }

void SetFunctionBailOutRecord(FunctionBailOutRecord \* record) { Assert(!HasFunctionBailOutRecord()); functionBailOutRecord = record; }

void RecordNativeThrowMap(SmallSpanSequenceIter& iter, uint32 offset, uint32 statementIndex, EntryPointInfo\* entryPoint, uint loopNum);

void RecordNativeBaseAddress(BYTE\* baseAddress, ptrdiff\_t codeSizeS, NativeCodeData \* data, NativeCodeData \* transferData, CodeGenNumberChunk \* numberChunks,

EntryPointInfo\* info, uint loopNum);

void SetNativeThrowSpanSequence(SmallSpanSequence \*seq, uint loopNum, LoopEntryPointInfo\* entryPoint);

BOOL GetMatchingStatementMapFromNativeAddress(DWORD\_PTR codeAddress, StatementData &data, uint loopNum, FunctionBody \*inlinee = nullptr);

BOOL GetMatchingStatementMapFromNativeOffset(DWORD\_PTR codeAddress, uint32 offset, StatementData &data, uint loopNum, FunctionBody \*inlinee = nullptr);

FunctionEntryPointInfo \* GetEntryPointFromNativeAddress(DWORD\_PTR codeAddress);

LoopEntryPointInfo \* GetLoopEntryPointInfoFromNativeAddress(DWORD\_PTR codeAddress, uint loopNum) const;

#endif

void InsertSymbolToRegSlotList(JsUtil::CharacterBuffer<WCHAR> const& propName, RegSlot reg, RegSlot totalRegsCount);

void InsertSymbolToRegSlotList(RegSlot reg, PropertyId propertyId, RegSlot totalRegsCount);

void SetPropertyIdsOfFormals(PropertyIdArray \* formalArgs);

bool DontRethunkAfterBailout() const { return dontRethunkAfterBailout; }

void SetDontRethunkAfterBailout() { dontRethunkAfterBailout = true; }

void ClearDontRethunkAfterBailout() { dontRethunkAfterBailout = false; }

void SaveState(ParseNodePtr pnode);

void RestoreState(ParseNodePtr pnode);

// Used for the debug purpose, this info will be stored (in the non-debug mode), when a function has all locals marked as non-local-referenced.

// So when we got to no-refresh debug mode, and try to re-use the same function body we can then enforce all locals to be non-local-referenced.

bool HasAllNonLocalReferenced() const { return m\_hasAllNonLocalReferenced; }

void SetAllNonLocalReferenced(bool set) { m\_hasAllNonLocalReferenced = set; }

bool HasSetIsObject() const { return m\_hasSetIsObject; }

void SetHasSetIsObject(bool set) { m\_hasSetIsObject = set; }

bool HasFuncExprNameReference() const { return m\_hasFunExprNameReference; }

void SetFuncExprNameReference(bool value) { m\_hasFunExprNameReference = value; }

bool GetChildCallsEval() const { return m\_ChildCallsEval; }

void SetChildCallsEval(bool value) { m\_ChildCallsEval = value; }

bool GetCallsEval() const { return m\_CallsEval; }

void SetCallsEval(bool set) { m\_CallsEval = set; }

bool HasReferenceableBuiltInArguments() const { return m\_hasReferenceableBuiltInArguments; }

void SetHasReferenceableBuiltInArguments(bool value) { m\_hasReferenceableBuiltInArguments = value; }

// Used for the debug purpose. This is to avoid setting all locals to non-local-referenced, multiple time for each child function.

bool HasDoneAllNonLocalReferenced() const { return m\_hasDoneAllNonLocalReferenced; }

void SetHasDoneAllNonLocalReferenced(bool set) { m\_hasDoneAllNonLocalReferenced = set; }

// Once the function compiled is sent m\_hasFunctionCompiledSent will be set to 'true'. The below check will be used only to determine during ProfileModeDeferredParse function.

bool HasFunctionCompiledSent() const { return m\_hasFunctionCompiledSent; }

void SetHasFunctionCompiledSent(bool set) { m\_hasFunctionCompiledSent = set; }

#if DBG\_DUMP

void DumpStatementMaps();

void Dump();

void PrintStatementSourceLine(uint statementIndex);

void PrintStatementSourceLineFromStartOffset(uint cchStartOffset);

void DumpScopes();

#endif

uint GetStatementStartOffset(const uint statementIndex);

#ifdef IR\_VIEWER

void GetSourceLineFromStartOffset(const uint startOffset, LPCUTF8 \*sourceBegin, LPCUTF8 \*sourceEnd,

ULONG \* line, LONG \* col);

void GetStatementSourceInfo(const uint statementIndex, LPCUTF8 \*sourceBegin, LPCUTF8 \*sourceEnd,

ULONG \* line, LONG \* col);

#endif

HRESULT RegisterFunction(BOOL fChangeMode, BOOL fOnlyCurrent = FALSE);

HRESULT ReportScriptCompiled();

HRESULT ReportFunctionCompiled();

void SetEntryToProfileMode();

void CheckAndRegisterFuncToDiag(ScriptContext \*scriptContext);

void SetEntryToDeferParseForDebugger();

void ResetEntryPoint();

void CleanupToReparse();

void AddDeferParseAttribute();

void RemoveDeferParseAttribute();

#if DBG

void MustBeInDebugMode();

#endif

static bool IsDummyGlobalRetStatement(const regex::Interval \*sourceSpan)

{

Assert(sourceSpan != nullptr);

return sourceSpan->begin == 0 && sourceSpan->end == 0;

}

static void GetShortNameFromUrl(\_\_in LPCWSTR pchUrl, \_Out\_writes\_z\_(cchBuffer) LPWSTR pchShortName, \_\_in size\_t cchBuffer);

template<class Fn>

void MapLoopHeaders(Fn fn) const

{

if(this->loopHeaderArray)

{

for(uint i = 0; i < loopCount; i++)

{

fn(i , &this->loopHeaderArray[i]);

}

}

}

template <class Fn>

void MapEntryPoints(Fn fn) const

{

if (this->entryPoints)

{

this->entryPoints->Map([&fn] (int index, RecyclerWeakReference<FunctionEntryPointInfo>\* entryPoint) {

FunctionEntryPointInfo\* strongRef = entryPoint->Get();

if (strongRef)

{

fn(index, strongRef);

}

});

}

}

bool DoJITLoopBody() const

{

return IsJitLoopBodyPhaseEnabled() && this->loopHeaderArray != nullptr;

}

bool ForceJITLoopBody() const

{

return IsJitLoopBodyPhaseForced() && !this->GetHasTry();

}

bool IsGeneratorAndJitIsDisabled()

{

return this->IsGenerator() && !(CONFIG\_ISENABLED(Js::JitES6GeneratorsFlag) && !this->GetHasTry());

}

FunctionBodyFlags \* GetAddressOfFlags() { return &this->flags; }

Js::RegSlot GetRestParamRegSlot();

public:

void RecordConstant(RegSlot location, Var var);

private:

inline void CheckEmpty();

inline void CheckNotExecuting();

SmallSpanSequence \*GetThrowSpanSequence(DWORD\_PTR codeAddress, uint loopNum);

BOOL GetMatchingStatementMap(StatementData &data, int statementIndex, FunctionBody \*inlinee);

#if ENABLE\_NATIVE\_CODEGEN

int GetStatementIndexFromNativeOffset(SmallSpanSequence \*pThrowSpanSequence, uint32 nativeOffset);

int GetStatementIndexFromNativeAddress(SmallSpanSequence \*pThrowSpanSequence, DWORD\_PTR codeAddress, DWORD\_PTR nativeBaseAddress);

#endif

void EnsureAuxStatementData();

StatementAdjustmentRecordList\* GetStatementAdjustmentRecords();

#ifdef DYNAMIC\_PROFILE\_MUTATOR

friend class DynamicProfileMutator;

friend class DynamicProfileMutatorImpl;

#endif

friend class RemoteFunctionBody;

};

typedef SynchronizableList<FunctionBody\*, JsUtil::List<FunctionBody\*, ArenaAllocator, false, Js::FreeListedRemovePolicy> > FunctionBodyList;

struct ScopeSlots

{

public:

static uint const MaxEncodedSlotCount = USHORT\_MAX;

// The slot index is at the same location as the vtable, so that we can distinguish between scope slot and frame display

static uint const EncodedSlotCountSlotIndex = 0;

static uint const ScopeMetadataSlotIndex = 1; // Either a FunctionBody\* or DebuggerScope\*

static uint const FirstSlotIndex = 2;

public:

ScopeSlots(Var\* slotArray) : slotArray(slotArray)

{

}

bool IsFunctionScopeSlotArray()

{

return FunctionBody::Is(slotArray[ScopeMetadataSlotIndex]);

}

FunctionBody\* GetFunctionBody()

{

Assert(IsFunctionScopeSlotArray());

return (FunctionBody\*)(slotArray[ScopeMetadataSlotIndex]);

}

DebuggerScope\* GetDebuggerScope()

{

Assert(!IsFunctionScopeSlotArray());

return (DebuggerScope\*)(slotArray[ScopeMetadataSlotIndex]);

}

Var GetScopeMetadataRaw() const

{

return slotArray[ScopeMetadataSlotIndex];

}

void SetScopeMetadata(Var scopeMetadataObj)

{

slotArray[ScopeMetadataSlotIndex] = scopeMetadataObj;

}

uint GetCount() const

{

return ::Math::PointerCastToIntegralTruncate<uint>(slotArray[EncodedSlotCountSlotIndex]);

}

void SetCount(uint count)

{

slotArray[EncodedSlotCountSlotIndex] = (Var)min<uint>(count, ScopeSlots::MaxEncodedSlotCount);

}

Var Get(uint i) const

{

Assert(i < GetCount());

return slotArray[i + FirstSlotIndex];

}

void Set(uint i, Var value)

{

Assert(i < GetCount());

slotArray[i + FirstSlotIndex] = value;

}

template<class Fn>

void Map(Fn fn)

{

uint count = GetCount();

for(uint i = 0; i < count; i++)

{

fn(GetSlot[i]);

}

}

// The first pointer sized value in the object for scope slots is the count, while it is a vtable

// for Activation object or with scope (a recyclable object)

// VTable values are always > 64K because they are a pointer, hence anything less than that implies

// a slot array.

// CONSIDER: Use TaggedInt instead of range of slot count to distinguish slot array with others.

static bool Is(void\* object)

{

size\_t slotCount = \*((size\_t\*)object);

if(slotCount <= MaxEncodedSlotCount)

{

return true;

}

return false;

}

private:

Var\* slotArray;

};

enum ScopeType

{

ScopeType\_ActivationObject,

ScopeType\_SlotArray,

ScopeType\_WithScope

};

// A FrameDisplay encodes a FunctionObject's scope chain. It is an array of scopes, where each scope can be either an inline slot array

// or a RecyclableObject. A FrameDisplay for a given FunctionObject will consist of the FrameDisplay from it's enclosing scope, with any additional

// scopes prepended. Due to with statements etc. a function may introduce multiple scopes to the FrameDisplay.

struct FrameDisplay

{

FrameDisplay(uint16 len, bool strictMode = false) :

tag(true),

length(len),

strictMode(strictMode)

#if \_M\_X64

, unused(0)

#endif

{

}

void SetTag(bool tag) { this->tag = tag; }

void SetItem(uint index, void\* item);

void \*GetItem(uint index);

uint16 GetLength() const { return length; }

void SetLength(uint16 len) { this->length = len; }

bool GetStrictMode() const { return strictMode; }

void SetStrictMode(bool flag) { this->strictMode = flag; }

void\*\* GetDataAddress() { return (void\*\*)&this->scopes; }

static uint32 GetOffsetOfStrictMode() { return offsetof(FrameDisplay, strictMode); }

static uint32 GetOffsetOfLength() { return offsetof(FrameDisplay, length); }

static uint32 GetOffsetOfScopes() { return offsetof(FrameDisplay, scopes); }

static ScopeType GetScopeType(void\* scope);

private:

bool tag; // Tag it so that the NativeCodeGenerator::IsValidVar would not think this is var

bool strictMode;

uint16 length;

#if defined(\_M\_X64\_OR\_ARM64)

uint32 unused;

#endif

void\* scopes[];

};

#pragma region Function Body helper classes

#pragma region Debugging related source classes

// Contains only the beginning part of the statement. This will mainly used in SmallSpanSequence which will further be compressed

// and stored in the buffer

struct StatementData

{

StatementData()

: sourceBegin(0),

bytecodeBegin(0)

{

}

int sourceBegin;

int bytecodeBegin;

};

struct StatementLocation

{

Js::FunctionBody\* function;

regex::Interval statement;

regex::Interval bytecodeSpan;

};

// Small span in the Statement buffer of the SmallSpanSequence

struct SmallSpan

{

ushort sourceBegin;

ushort bytecodeBegin;

SmallSpan(uint32 val)

{

sourceBegin = (ushort)(val >> 16);

bytecodeBegin = (ushort)(val & 0x0000FFFF);

}

operator unsigned int()

{

return (uint32)sourceBegin << 16 | bytecodeBegin;

}

};

// Iterator which contains the state at particular index. These values will used when fetching next item from

// SmallSpanSequence

class SmallSpanSequenceIter

{

friend class SmallSpanSequence;

public:

SmallSpanSequenceIter()

: accumulatedIndex(-1),

accumulatedSourceBegin(0),

accumulatedBytecodeBegin(0),

indexOfActualOffset(0)

{

}

// Below are used for fast access when the last access happened nearby.

// so the actual index would be accumulatedIndex / 2 + (remainder for which byte).

int accumulatedIndex;

int accumulatedSourceBegin;

int accumulatedBytecodeBegin;

int indexOfActualOffset;

};

// This class compacts the range of the statement to BYTEs instead of ints.

// Instead of having start and end as int32s we will have them stored in bytes, and they will be

// treated as start offset and end offset.

// For simplicity, this class should be heap allocated, since it can be allocated from either the background

// or main thread.

class SmallSpanSequence

{

friend class SmallSpanSequenceIter;

friend class ByteCodeBufferBuilder;

friend class ByteCodeBufferReader;

private:

// Each item in the list contains two set of begins (one for bytecode and for sourcespan).

// The allowed valued for source and bytecode span is in between SHORT\_MAX - 1 to SHORT\_MIN (inclusive).

// otherwise its a miss

JsUtil::GrowingUint32HeapArray \* pStatementBuffer;

// Contains list of values which are missed in StatementBuffer.

JsUtil::GrowingUint32HeapArray \* pActualOffsetList;

// The first value of the sequence

int baseValue;

BOOL GetRangeAt(int index, SmallSpanSequenceIter &iter, int \* pCountOfMissed, StatementData & data);

ushort GetDiff(int current, int prev);

public:

SmallSpanSequence();

~SmallSpanSequence()

{

Cleanup();

}

void Cleanup()

{

if (pStatementBuffer != nullptr)

{

HeapDelete(pStatementBuffer);

}

if (pActualOffsetList != nullptr)

{

HeapDelete(pActualOffsetList);

}

}

// Trys to match passed bytecode in the statement, and returns the statement which includes that.

BOOL GetMatchingStatementFromBytecode(int bytecode, SmallSpanSequenceIter &iter, StatementData & data);

// Record the statement data in the statement buffer in the compressed manner.

BOOL RecordARange(SmallSpanSequenceIter &iter, StatementData \* data);

// Reset the accumulator's state and value.

void Reset(SmallSpanSequenceIter &iter);

uint32 Count() const { return pStatementBuffer ? pStatementBuffer->Count() : 0; }

BOOL Item(int index, SmallSpanSequenceIter &iter, StatementData &data);

// Below function will not change any state, so it will not alter accumulated index and value

BOOL Seek(int index, StatementData & data);

SmallSpanSequence \* Clone();

};

#pragma endregion

// This container represent the property ids for the locals which are placed at direct slot

// and list of formals args if user has not used the arguments object in the script for the current function

struct PropertyIdOnRegSlotsContainer

{

PropertyId \* propertyIdsForRegSlots;

uint length;

PropertyIdArray \* propertyIdsForFormalArgs;

PropertyIdOnRegSlotsContainer();

static PropertyIdOnRegSlotsContainer \* New(Recycler \* recycler);

void CreateRegSlotsArray(Recycler \* recycler, uint \_length);

void SetFormalArgs(PropertyIdArray \* formalArgs);

// Helper methods

void Insert(RegSlot reg, PropertyId propId);

void FetchItemAt(uint index, FunctionBody \*pFuncBody, \_\_out PropertyId \*pPropId, \_\_out RegSlot \*pRegSlot);

// Whether reg belongs to non-temp locals

bool IsRegSlotFormal(RegSlot reg);

};

// Flags for the DebuggerScopeProperty object.

typedef int DebuggerScopePropertyFlags;

const int DebuggerScopePropertyFlags\_None = 0x000000000;

const int DebuggerScopePropertyFlags\_Const = 0x000000001;

const int DebuggerScopePropertyFlags\_CatchObject = 0x000000002;

const int DebuggerScopePropertyFlags\_WithObject = 0x000000004;

const int DebuggerScopePropertyFlags\_ForInOrOfCollection = 0x000000008;

// Used to store local property info for with/catch objects, lets, or consts

// that are needed for the debugger.

class DebuggerScopeProperty

{

public:

Js::PropertyId propId; // The property ID of the scope variable.

RegSlot location; // Contains the location of the scope variable (regslot, slotarray, direct).

int byteCodeInitializationOffset; // The byte code offset used when comparing let/const variables for dead zone exclusion debugger side.

DebuggerScopePropertyFlags flags; // Flags for the property.

bool IsConst() const { return (flags & DebuggerScopePropertyFlags\_Const) != 0; }

bool IsCatchObject() const { return (flags & DebuggerScopePropertyFlags\_CatchObject) != 0; }

bool IsWithObject() const { return (flags & DebuggerScopePropertyFlags\_WithObject) != 0; }

bool IsForInOrForOfCollectionScope() const { return (flags & DebuggerScopePropertyFlags\_ForInOrOfCollection) != 0; }

public:

// Determines if the current property is in a dead zone. Note that the property makes

// no assumptions about what scope it's in, that is determined by DebuggerScope.

// byteCodeOffset - The current offset in bytecode that the debugger is at.

bool IsInDeadZone(int byteCodeOffset) const

{

if (IsForInOrForOfCollectionScope())

{

// These are let/const loop variables of a for-in or for-of loop

// in the scope for the collection expression. They are always

// in TDZ in this scope, never initialized by the bytecode.

return true;

}

if (this->byteCodeInitializationOffset == Constants::InvalidByteCodeOffset && !(IsCatchObject() || IsWithObject()))

{

AssertMsg(false, "Debug let/const property never had its initialization point updated. This indicates that a Ld or St operation in ByteCodeGenerator was missed that needs to have DebuggerScope::UpdatePropertyInitializationOffset() added to it.");

return false;

}

return byteCodeOffset < this->byteCodeInitializationOffset;

}

};

// Used to track with, catch, and block scopes for the debugger to determine context.

class DebuggerScope

{

public:

typedef JsUtil::List<DebuggerScopeProperty> DebuggerScopePropertyList;

DebuggerScope(Recycler\* recycler, DiagExtraScopesType scopeType, RegSlot scopeLocation, int rangeBegin)

: scopeType(scopeType),

scopeProperties(nullptr),

parentScope(nullptr),

siblingScope(nullptr),

scopeLocation(scopeLocation),

recycler(recycler)

{

this->range.begin = rangeBegin;

this->range.end = -1;

}

DebuggerScope \* GetSiblingScope(RegSlot location, FunctionBody \*functionBody);

void AddProperty(RegSlot location, Js::PropertyId propertyId, DebuggerScopePropertyFlags flags);

bool GetPropertyIndex(Js::PropertyId propertyId, int& i);

bool IsOffsetInScope(int offset) const;

bool Contains(Js::PropertyId propertyId, RegSlot location) const;

bool IsBlockScope() const;

bool IsBlockObjectScope() const

{

return this->scopeType == Js::DiagBlockScopeInObject;

}

bool IsCatchScope() const;

bool IsWithScope() const;

bool IsSlotScope() const;

bool HasProperties() const;

bool IsAncestorOf(const DebuggerScope\* potentialChildScope);

bool AreAllPropertiesInDeadZone(int byteCodeOffset) const;

RegSlot GetLocation() const { Assert(IsOwnScope()); return scopeLocation; }

bool IsOwnScope() const { return scopeLocation != Js::Constants::NoRegister; }

bool TryGetProperty(Js::PropertyId propertyId, RegSlot location, DebuggerScopeProperty\* outScopeProperty) const;

bool TryGetValidProperty(Js::PropertyId propertyId, RegSlot location, int offset, DebuggerScopeProperty\* outScopeProperty, bool\* isInDeadZone) const;

bool UpdatePropertyInitializationOffset(RegSlot location, Js::PropertyId propertyId, int byteCodeOffset, bool isFunctionDeclaration = false);

void UpdateDueToByteCodeRegeneration(DiagExtraScopesType scopeType, int start, RegSlot scopeLocation);

void UpdatePropertiesInForInOrOfCollectionScope();

void SetParentScope(DebuggerScope\* parentScope) { this->parentScope = parentScope; }

DebuggerScope\* GetParentScope() const { return parentScope; }

DebuggerScope\* FindCommonAncestor(DebuggerScope\* debuggerScope);

int GetEnd() const { return range.end; }

int GetStart() const { return range.begin; }

void SetScopeLocation(RegSlot scopeLocation) { this->scopeLocation = scopeLocation; }

void SetBegin(int begin);

void SetEnd(int end);

#if DBG

void Dump();

PCWSTR GetDebuggerScopeTypeString(DiagExtraScopesType scopeType);

#endif

public:

// The list of scope properties in this scope object.

// For with scope: Has 1 property that represents the scoped object.

// For catch scope: Has 1 property that represents the exception object.

// For block scope: Has 0-n properties that represent let/const variables in that scope.

DebuggerScopePropertyList\* scopeProperties;

DiagExtraScopesType scopeType; // The type of scope being represented (With, Catch, or Block scope).

DebuggerScope\* siblingScope; // Valid only when current scope is slot/activationobject and symbols are on direct regslot

static const int InvalidScopeIndex = -1;

private:

int GetScopeDepth() const;

bool UpdatePropertyInitializationOffsetInternal(RegSlot location, Js::PropertyId propertyId, int byteCodeOffset, bool isFunctionDeclaration = false);

void EnsurePropertyListIsAllocated();

private:

DebuggerScope\* parentScope;

regex::Interval range; // The start and end byte code writer offsets used when comparing where the debugger is currently stopped at (breakpoint location).

RegSlot scopeLocation;

Recycler\* recycler;

};

class ScopeObjectChain

{

public:

typedef JsUtil::List<DebuggerScope\*> ScopeObjectChainList;

ScopeObjectChain(Recycler\* recycler)

: pScopeChain(nullptr)

{

pScopeChain = RecyclerNew(recycler, ScopeObjectChainList, recycler);

}

// This function will return DebuggerScopeProperty when the property is found and correctly in the range.

// If the property is found, but the scope is not in the range, it will return false, but the out param (isPropertyInDebuggerScope) will set to true,

// and isConst will be updated.

// If the property is not found at all, it will return false, and isPropertyInDebuggerScope will be false.

bool TryGetDebuggerScopePropertyInfo(PropertyId propertyId, RegSlot location, int offset, bool\* isPropertyInDebuggerScope, bool \*isConst, bool\* isInDeadZone);

// List of all Scope Objects in a function. Scopes are added to this list as when they are created in bytecode gen part.

ScopeObjectChainList\* pScopeChain;

};

#pragma endregion

} // 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 "RuntimeBasePch.h"

#include "ByteCode\ByteCodeAPI.h"

#include "ByteCode\ByteCodeDumper.h"

#include "Language\AsmJsTypes.h"

#include "Language\AsmJsModule.h"

#include "ByteCode\ByteCodeSerializer.h"

#include "Language\FunctionCodeGenRuntimeData.h"

#include "ByteCode\ScopeInfo.h"

#include "Base\EtwTrace.h"

#ifdef DYNAMIC\_PROFILE\_MUTATOR

#include "Language\DynamicProfileMutator.h"

#endif

#include "Language\SourceDynamicProfileManager.h"

#include "Debug\ProbeContainer.h"

#include "Debug\DebugContext.h"

#include "Parser.h"

#include "RegexCommon.h"

#include "RegexPattern.h"

#include "Library\RegexHelper.h"

#include "Language\InterpreterStackFrame.h"

#include "Library\ModuleRoot.h"

#include "Types\PathTypeHandler.h"

namespace Js

{

#ifdef FIELD\_ACCESS\_STATS

void FieldAccessStats::Add(FieldAccessStats\* other)

{

Assert(other != nullptr);

this->totalInlineCacheCount += other->totalInlineCacheCount;

this->noInfoInlineCacheCount += other->noInfoInlineCacheCount;

this->monoInlineCacheCount += other->monoInlineCacheCount;

this->emptyMonoInlineCacheCount += other->emptyMonoInlineCacheCount;

this->polyInlineCacheCount += other->polyInlineCacheCount;

this->nullPolyInlineCacheCount += other->nullPolyInlineCacheCount;

this->emptyPolyInlineCacheCount += other->emptyPolyInlineCacheCount;

this->ignoredPolyInlineCacheCount += other->ignoredPolyInlineCacheCount;

this->highUtilPolyInlineCacheCount += other->highUtilPolyInlineCacheCount;

this->lowUtilPolyInlineCacheCount += other->lowUtilPolyInlineCacheCount;

this->equivPolyInlineCacheCount += other->equivPolyInlineCacheCount;

this->nonEquivPolyInlineCacheCount += other->nonEquivPolyInlineCacheCount;

this->disabledPolyInlineCacheCount += other->disabledPolyInlineCacheCount;

this->clonedMonoInlineCacheCount += other->clonedMonoInlineCacheCount;

this->clonedPolyInlineCacheCount += other->clonedPolyInlineCacheCount;

}

#endif

// FunctionProxy methods

FunctionProxy::FunctionProxy(JavascriptMethod entryPoint, Attributes attributes, int nestedCount, int derivedSize, LocalFunctionId functionId, ScriptContext\* scriptContext, Utf8SourceInfo\* utf8SourceInfo, uint functionNumber):

FunctionInfo(entryPoint, attributes, functionId, (FunctionBody\*) this),

m\_nestedCount(nestedCount),

m\_isTopLevel(false),

m\_isPublicLibraryCode(false),

m\_derivedSize(derivedSize),

m\_scriptContext(scriptContext),

m\_utf8SourceInfo(utf8SourceInfo),

m\_referenceInParentFunction(nullptr),

m\_functionNumber(functionNumber),

m\_defaultEntryPointInfo(nullptr),

m\_functionObjectTypeList(nullptr)

{

PERF\_COUNTER\_INC(Code, TotalFunction);

}

uint FunctionProxy::GetSourceContextId() const

{

return m\_utf8SourceInfo->GetSrcInfo()->sourceContextInfo->sourceContextId;

}

wchar\_t\* FunctionProxy::GetDebugNumberSet(wchar(&bufferToWriteTo)[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE]) const

{

// (#%u.%u), #%u --> (source file Id . function Id) , function Number

int len = swprintf\_s(bufferToWriteTo, MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE, L" (#%d.%u), #%u",

(int)this->GetSourceContextId(), this->GetLocalFunctionId(), this->GetFunctionNumber());

Assert(len > 8);

return bufferToWriteTo;

}

bool

FunctionProxy::IsFunctionBody() const

{

return !IsDeferredDeserializeFunction() && GetParseableFunctionInfo()->IsFunctionParsed();

}

uint

ParseableFunctionInfo::GetSourceIndex() const

{

return this->m\_sourceIndex;

}

LPCUTF8

ParseableFunctionInfo::GetSource(const wchar\_t\* reason) const

{

return this->m\_utf8SourceInfo->GetSource(reason == nullptr ? L"ParseableFunctionInfo::GetSource" : reason) + this->StartOffset();

}

LPCUTF8

ParseableFunctionInfo::GetStartOfDocument(const wchar\_t\* reason) const

{

return this->m\_utf8SourceInfo->GetSource(reason == nullptr ? L"ParseableFunctionInfo::GetStartOfDocument" : reason);

}

bool

ParseableFunctionInfo::IsDynamicFunction() const

{

return this->m\_isDynamicFunction;

}

bool

ParseableFunctionInfo::IsDynamicScript() const

{

return this->GetSourceContextInfo()->IsDynamic();

}

charcount\_t

ParseableFunctionInfo::StartInDocument() const

{

return this->m\_cchStartOffset;

}

uint

ParseableFunctionInfo::StartOffset() const

{

return this->m\_cbStartOffset;

}

void ParseableFunctionInfo::RegisterFuncToDiag(ScriptContext \* scriptContext, wchar\_t const \* pszTitle)

{

// Register the function to the PDM as eval code (the debugger app will show file as 'eval code')

scriptContext->GetDebugContext()->RegisterFunction(this, pszTitle);

}

// Given an offset into the source buffer, determine if the end of this SourceInfo

// lies after the given offset.

bool

FunctionBody::EndsAfter(size\_t offset) const

{

return offset < this->StartOffset() + this->LengthInBytes();

}

void

FunctionBody::RecordStatementMap(StatementMap\* pStatementMap)

{

Assert(!this->m\_sourceInfo.pSpanSequence);

Recycler\* recycler = this->m\_scriptContext->GetRecycler();

StatementMapList \* statementMaps = this->GetStatementMaps();

if (!statementMaps)

{

statementMaps = RecyclerNew(recycler, StatementMapList, recycler);

this->pStatementMaps = statementMaps;

}

statementMaps->Add(pStatementMap);

}

void

FunctionBody::RecordStatementMap(SmallSpanSequenceIter &iter, StatementData \* data)

{

Assert(!this->GetStatementMaps());

if (!this->m\_sourceInfo.pSpanSequence)

{

this->m\_sourceInfo.pSpanSequence = HeapNew(SmallSpanSequence);

}

this->m\_sourceInfo.pSpanSequence->RecordARange(iter, data);

}

void

FunctionBody::RecordStatementAdjustment(uint offset, StatementAdjustmentType adjType)

{

this->EnsureAuxStatementData();

Recycler\* recycler = this->m\_scriptContext->GetRecycler();

if (this->GetStatementAdjustmentRecords() == nullptr)

{

m\_sourceInfo.m\_auxStatementData->m\_statementAdjustmentRecords = RecyclerNew(recycler, StatementAdjustmentRecordList, recycler);

}

StatementAdjustmentRecord record(adjType, offset);

this->GetStatementAdjustmentRecords()->Add(record); // Will copy stack value and put the copy into the container.

}

BOOL

FunctionBody::GetBranchOffsetWithin(uint start, uint end, StatementAdjustmentRecord\* record)

{

Assert(start < end);

if (!this->GetStatementAdjustmentRecords())

{

// No Offset

return FALSE;

}

int count = this->GetStatementAdjustmentRecords()->Count();

for (int i = 0; i < count; i++)

{

StatementAdjustmentRecord item = this->GetStatementAdjustmentRecords()->Item(i);

if (item.GetByteCodeOffset() > start && item.GetByteCodeOffset() < end)

{

\*record = item;

return TRUE;

}

}

// No offset found in the range.

return FALSE;

}

ScriptContext\* EntryPointInfo::GetScriptContext()

{

Assert(!IsCleanedUp());

return this->library->GetScriptContext();

}

#if DBG\_DUMP | defined(VTUNE\_PROFILING)

void

EntryPointInfo::RecordNativeMap(uint32 nativeOffset, uint32 statementIndex)

{

int count = nativeOffsetMaps.Count();

if (count)

{

NativeOffsetMap\* previous = &nativeOffsetMaps.Item(count-1);

// Check if the range is still not finished.

if (previous->nativeOffsetSpan.begin == previous->nativeOffsetSpan.end)

{

if (previous->statementIndex == statementIndex)

{

// If the statement index is the same, we can continue with the previous range

return;

}

// If the range is empty, replace the previous range.

if ((uint32)previous->nativeOffsetSpan.begin == nativeOffset)

{

if (statementIndex == Js::Constants::NoStatementIndex)

{

nativeOffsetMaps.RemoveAtEnd();

}

else

{

previous->statementIndex = statementIndex;

}

return;

}

// Close the previous range

previous->nativeOffsetSpan.end = nativeOffset;

}

}

if (statementIndex == Js::Constants::NoStatementIndex)

{

// We do not explicitly record the offsets that do not map to user code.

return;

}

NativeOffsetMap map;

map.statementIndex = statementIndex;

map.nativeOffsetSpan.begin = nativeOffset;

map.nativeOffsetSpan.end = nativeOffset;

nativeOffsetMaps.Add(map);

}

#endif

void

FunctionBody::CopySourceInfo(ParseableFunctionInfo\* originalFunctionInfo)

{

this->m\_sourceIndex = originalFunctionInfo->GetSourceIndex();

this->m\_cchStartOffset = originalFunctionInfo->StartInDocument();

this->m\_cchLength = originalFunctionInfo->LengthInChars();

this->m\_lineNumber = originalFunctionInfo->GetRelativeLineNumber();

this->m\_columnNumber = originalFunctionInfo->GetRelativeColumnNumber();

this->m\_isEval = originalFunctionInfo->IsEval();

this->m\_isDynamicFunction = originalFunctionInfo->IsDynamicFunction();

this->m\_cbStartOffset = originalFunctionInfo->StartOffset();

this->m\_cbLength = originalFunctionInfo->LengthInBytes();

this->FinishSourceInfo();

}

// When sourceInfo is complete, register this functionBody to utf8SourceInfo. This ensures we never

// put incomplete functionBody into utf8SourceInfo map. (Previously we do it in FunctionBody constructor.

// If an error occurs thereafter before SetSourceInfo, e.g. OOM, we'll have an incomplete functionBody

// in utf8SourceInfo map whose source range is unknown and can't be reparsed.)

void FunctionBody::FinishSourceInfo()

{

m\_utf8SourceInfo->SetFunctionBody(this);

}

RegSlot FunctionBody::GetFrameDisplayRegister() const

{

return this->m\_sourceInfo.frameDisplayRegister;

}

void FunctionBody::SetFrameDisplayRegister(RegSlot frameDisplayRegister)

{

this->m\_sourceInfo.frameDisplayRegister = frameDisplayRegister;

}

RegSlot FunctionBody::GetObjectRegister() const

{

return this->m\_sourceInfo.objectRegister;

}

void FunctionBody::SetObjectRegister(RegSlot objectRegister)

{

this->m\_sourceInfo.objectRegister = objectRegister;

}

ScopeObjectChain \*FunctionBody::GetScopeObjectChain() const

{

return this->m\_sourceInfo.pScopeObjectChain;

}

void FunctionBody::SetScopeObjectChain(ScopeObjectChain \*pScopeObjectChain)

{

this->m\_sourceInfo.pScopeObjectChain = pScopeObjectChain;

}

ByteBlock \*FunctionBody::GetProbeBackingBlock()

{

return this->m\_sourceInfo.m\_probeBackingBlock;

}

void FunctionBody::SetProbeBackingBlock(ByteBlock\* probeBackingBlock)

{

this->m\_sourceInfo.m\_probeBackingBlock = probeBackingBlock;

}

FunctionBody \* FunctionBody::NewFromRecycler(ScriptContext \* scriptContext, const wchar\_t \* displayName, uint displayNameLength, uint displayShortNameOffset, uint nestedCount,

Utf8SourceInfo\* sourceInfo, uint uScriptId, Js::LocalFunctionId functionId, Js::PropertyRecordList\* boundPropertyRecords, Attributes attributes

#ifdef PERF\_COUNTERS

, bool isDeserializedFunction

#endif

)

{

return FunctionBody::NewFromRecycler(scriptContext, displayName, displayNameLength, displayShortNameOffset, nestedCount, sourceInfo,

scriptContext->GetThreadContext()->NewFunctionNumber(), uScriptId, functionId, boundPropertyRecords, attributes

#ifdef PERF\_COUNTERS

, isDeserializedFunction

#endif

);

}

FunctionBody \* FunctionBody::NewFromRecycler(ScriptContext \* scriptContext, const wchar\_t \* displayName, uint displayNameLength, uint displayShortNameOffset, uint nestedCount,

Utf8SourceInfo\* sourceInfo, uint uFunctionNumber, uint uScriptId, Js::LocalFunctionId functionId, Js::PropertyRecordList\* boundPropertyRecords, Attributes attributes

#ifdef PERF\_COUNTERS

, bool isDeserializedFunction

#endif

)

{

#ifdef PERF\_COUNTERS

return RecyclerNewWithBarrierFinalizedPlus(scriptContext->GetRecycler(), nestedCount \* sizeof(FunctionBody\*), FunctionBody, scriptContext, displayName, displayNameLength, displayShortNameOffset, nestedCount, sourceInfo, uFunctionNumber, uScriptId, functionId, boundPropertyRecords, attributes, isDeserializedFunction);

#else

return RecyclerNewWithBarrierFinalizedPlus(scriptContext->GetRecycler(), nestedCount \* sizeof(FunctionBody\*), FunctionBody, scriptContext, displayName, displayNameLength, displayShortNameOffset, nestedCount, sourceInfo, uFunctionNumber, uScriptId, functionId, boundPropertyRecords, attributes);

#endif

}

FunctionBody::FunctionBody(ScriptContext\* scriptContext, const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, uint nestedCount,

Utf8SourceInfo\* utf8SourceInfo, uint uFunctionNumber, uint uScriptId,

Js::LocalFunctionId functionId, Js::PropertyRecordList\* boundPropertyRecords, Attributes attributes

#ifdef PERF\_COUNTERS

, bool isDeserializedFunction

#endif

) :

ParseableFunctionInfo(scriptContext->CurrentThunk, nestedCount, sizeof(FunctionBody), functionId, utf8SourceInfo, scriptContext, uFunctionNumber, displayName, displayNameLength, displayShortNameOffset, attributes, boundPropertyRecords),

m\_uScriptId(uScriptId),

m\_varCount(0),

m\_outParamMaxDepth(0),

m\_firstTmpReg(Constants::NoRegister),

loopCount(0),

cleanedUp(false),

sourceInfoCleanedUp(false),

profiledLdElemCount(0),

profiledStElemCount(0),

profiledCallSiteCount(0),

profiledArrayCallSiteCount(0),

profiledDivOrRemCount(0),

profiledSwitchCount(0),

profiledReturnTypeCount(0),

profiledSlotCount(0),

m\_isFuncRegistered(false),

m\_isFuncRegisteredToDiag(false),

m\_hasBailoutInstrInJittedCode(false),

m\_depth(0),

inlineDepth(0),

m\_pendingLoopHeaderRelease(false),

inlineCacheCount(0),

rootObjectLoadInlineCacheStart(0),

rootObjectStoreInlineCacheStart(0),

isInstInlineCacheCount(0),

objLiteralCount(0),

literalRegexCount(0),

innerScopeCount(0),

hasCachedScopePropIds(false),

m\_byteCodeCount(0),

m\_byteCodeWithoutLDACount(0),

m\_argUsedForBranch(0),

m\_byteCodeInLoopCount(0),

m\_envDepth((uint16)-1),

flags(Flags\_HasNoExplicitReturnValue),

m\_hasFinally(false),

#if ENABLE\_PROFILE\_INFO

dynamicProfileInfo(nullptr),

polymorphicCallSiteInfoHead(nullptr),

#endif

savedInlinerVersion(0),

#if ENABLE\_NATIVE\_CODEGEN

savedImplicitCallsFlags(ImplicitCall\_HasNoInfo),

#endif

savedPolymorphicCacheState(0),

functionBailOutRecord(nullptr),

hasExecutionDynamicProfileInfo(false),

m\_hasAllNonLocalReferenced(false),

m\_hasSetIsObject(false),

m\_hasFunExprNameReference(false),

m\_CallsEval(false),

m\_ChildCallsEval(false),

m\_hasReferenceableBuiltInArguments(false),

m\_firstFunctionObject(true),

m\_inlineCachesOnFunctionObject(false),

m\_hasDoneAllNonLocalReferenced(false),

m\_hasFunctionCompiledSent(false),

byteCodeCache(nullptr),

stackNestedFuncParent(nullptr),

localClosureRegister(Constants::NoRegister),

localFrameDisplayRegister(Constants::NoRegister),

envRegister(Constants::NoRegister),

thisRegisterForEventHandler(Constants::NoRegister),

firstInnerScopeRegister(Constants::NoRegister),

funcExprScopeRegister(Constants::NoRegister),

m\_tag(TRUE),

m\_nativeEntryPointUsed(FALSE),

debuggerScopeIndex(0),

bailOnMisingProfileCount(0),

bailOnMisingProfileRejitCount(0),

auxBlock(nullptr),

auxContextBlock(nullptr),

byteCodeBlock(nullptr),

entryPoints(nullptr),

loopHeaderArray(nullptr),

m\_constTable(nullptr),

literalRegexes(nullptr),

asmJsFunctionInfo(nullptr),

asmJsModuleInfo(nullptr),

m\_codeGenRuntimeData(nullptr),

m\_codeGenGetSetRuntimeData(nullptr),

pStatementMaps(nullptr),

inlineCaches(nullptr),

polymorphicInlineCachesHead(nullptr),

cacheIdToPropertyIdMap(nullptr),

referencedPropertyIdMap(nullptr),

propertyIdsForScopeSlotArray(nullptr),

propertyIdOnRegSlotsContainer(nullptr),

executionMode(ExecutionMode::Interpreter),

interpreterLimit(0),

autoProfilingInterpreter0Limit(0),

profilingInterpreter0Limit(0),

autoProfilingInterpreter1Limit(0),

simpleJitLimit(0),

profilingInterpreter1Limit(0),

fullJitThreshold(0),

fullJitRequeueThreshold(0),

committedProfiledIterations(0),

simpleJitEntryPointInfo(nullptr),

wasCalledFromLoop(false),

hasScopeObject(false),

hasNestedLoop(false),

recentlyBailedOutOfJittedLoopBody(false),

serializationIndex(-1),

m\_isAsmJsScheduledForFullJIT(false),

m\_asmJsTotalLoopCount(0),

//

// Even if the function does not require any locals, we must always have "R0" to propagate

// a return value. By enabling this here, we avoid unnecessary conditionals during execution.

//

m\_constCount(1)

#ifdef IR\_VIEWER

,m\_isIRDumpEnabled(false)

,m\_irDumpBaseObject(nullptr)

#endif /\* IR\_VIEWER \*/

, m\_isFromNativeCodeModule(false)

, interpretedCount(0)

, loopInterpreterLimit(CONFIG\_FLAG(LoopInterpretCount))

, hasHotLoop(false)

, m\_isPartialDeserializedFunction(false)

#ifdef PERF\_COUNTERS

, m\_isDeserializedFunction(isDeserializedFunction)

#endif

#if DBG

, m\_DEBUG\_executionCount(0)

, m\_nativeEntryPointIsInterpreterThunk(false)

, m\_canDoStackNestedFunc(false)

, m\_inlineCacheTypes(nullptr)

, m\_iProfileSession(-1)

, initializedExecutionModeAndLimits(false)

#endif

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

, regAllocLoadCount(0)

, regAllocStoreCount(0)

, callCountStats(0)

#endif

{

this->SetDefaultFunctionEntryPointInfo((FunctionEntryPointInfo\*) this->GetDefaultEntryPointInfo(), DefaultEntryThunk);

this->m\_hasBeenParsed = true;

#ifdef PERF\_COUNTERS

if (isDeserializedFunction)

{

PERF\_COUNTER\_INC(Code, DeserializedFunctionBody);

}

#endif

Assert(!utf8SourceInfo || m\_uScriptId == utf8SourceInfo->GetSrcInfo()->sourceContextInfo->sourceContextId);

// Sync entryPoints changes to etw rundown lock

CriticalSection\* syncObj = scriptContext->GetThreadContext()->GetEtwRundownCriticalSection();

this->entryPoints = RecyclerNew(this->m\_scriptContext->GetRecycler(), FunctionEntryPointList, this->m\_scriptContext->GetRecycler(), syncObj);

this->AddEntryPointToEntryPointList(this->GetDefaultFunctionEntryPointInfo());

Assert(this->GetDefaultEntryPointInfo()->address != nullptr);

InitDisableInlineApply();

InitDisableInlineSpread();

}

void FunctionBody::SetDefaultFunctionEntryPointInfo(FunctionEntryPointInfo\* entryPointInfo, const JavascriptMethod originalEntryPoint)

{

Assert(entryPointInfo);

// Need to set twice since ProxyEntryPointInfo cast points to an interior pointer

this->m\_defaultEntryPointInfo = (ProxyEntryPointInfo\*) entryPointInfo;

this->defaultFunctionEntryPointInfo = entryPointInfo;

SetOriginalEntryPoint(originalEntryPoint);

}

ByteBlock\*

FunctionBody::GetAuxiliaryData()

{

return this->auxBlock;

}

ByteBlock\*

FunctionBody::GetAuxiliaryContextData()

{

return this->auxContextBlock;

}

ByteBlock\*

FunctionBody::GetByteCode()

{

return this->byteCodeBlock;

}

// Returns original bytecode without probes (such as BPs).

ByteBlock\*

FunctionBody::GetOriginalByteCode()

{

if (m\_sourceInfo.m\_probeBackingBlock)

{

return m\_sourceInfo.m\_probeBackingBlock;

}

else

{

return this->GetByteCode();

}

}

const ByteCodeCache \*

FunctionBody::GetByteCodeCache() const

{

return byteCodeCache;

}

const int

FunctionBody::GetSerializationIndex() const

{

return serializationIndex;

}

const wchar\_t\* ParseableFunctionInfo::GetExternalDisplayName() const

{

return GetExternalDisplayName(this);

}

RegSlot

FunctionBody::GetLocalsCount()

{

return m\_constCount + m\_varCount;

}

RegSlot

FunctionBody::GetVarCount()

{

return m\_varCount;

}

// Returns the number of non-temp local vars.

uint32

FunctionBody::GetNonTempLocalVarCount()

{

Assert(this->GetEndNonTempLocalIndex() >= this->GetFirstNonTempLocalIndex());

return this->GetEndNonTempLocalIndex() - this->GetFirstNonTempLocalIndex();

}

uint32

FunctionBody::GetFirstNonTempLocalIndex()

{

// First local var starts when the const vars end.

return m\_constCount;

}

uint32

FunctionBody::GetEndNonTempLocalIndex()

{

// It will give the index on which current non temp locals ends, which is a first temp reg.

return m\_firstTmpReg != Constants::NoRegister ? m\_firstTmpReg : GetLocalsCount();

}

bool

FunctionBody::IsNonTempLocalVar(uint32 varIndex)

{

return GetFirstNonTempLocalIndex() <= varIndex && varIndex < GetEndNonTempLocalIndex();

}

bool

FunctionBody::GetSlotOffset(RegSlot slotId, int32 \* slotOffset, bool allowTemp)

{

if (IsNonTempLocalVar(slotId) || allowTemp)

{

\*slotOffset = (slotId - GetFirstNonTempLocalIndex()) \* DIAGLOCALSLOTSIZE;

return true;

}

return false;

}

void

FunctionBody::SetConstantCount(

RegSlot cNewConstants) // New register count

{

CheckNotExecuting();

AssertMsg(m\_constCount <= cNewConstants, "Cannot shrink register usage");

m\_constCount = cNewConstants;

}

void

FunctionBody::SetVarCount(

RegSlot cNewVars) // New register count

{

CheckNotExecuting();

AssertMsg(m\_varCount <= cNewVars, "Cannot shrink register usage");

m\_varCount = cNewVars;

}

RegSlot

FunctionBody::GetYieldRegister()

{

return GetEndNonTempLocalIndex() - 1;

}

RegSlot

FunctionBody::GetFirstTmpReg()

{

AssertMsg(m\_firstTmpReg != Constants::NoRegister, "First temp hasn't been set yet");

return m\_firstTmpReg;

}

void

FunctionBody::SetFirstTmpReg(

RegSlot firstTmpReg)

{

CheckNotExecuting();

AssertMsg(m\_firstTmpReg == Constants::NoRegister, "Should not be resetting the first temp");

m\_firstTmpReg = firstTmpReg;

}

RegSlot

FunctionBody::GetTempCount()

{

return GetLocalsCount() - m\_firstTmpReg;

}

void

FunctionBody::SetOutParamDepth(

RegSlot cOutParamsDepth)

{

CheckNotExecuting();

m\_outParamMaxDepth = cOutParamsDepth;

}

RegSlot

FunctionBody::GetOutParamsDepth()

{

return m\_outParamMaxDepth;

}

ModuleID

FunctionBody::GetModuleID() const

{

return this->GetHostSrcInfo()->moduleID;

}

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

///

/// FunctionBody::BeginExecution

///

/// BeginExecution() is called by InterpreterStackFrame when a function begins execution.

/// - Once started execution, the function may not be modified, as it would

/// change the stack-frame layout:

/// - This is a debug-only check because of the runtime cost. At release time,

/// a stack-walk will be performed by GC to determine which functions are

/// executing.

///

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

void

FunctionBody::BeginExecution()

{

#if DBG

m\_DEBUG\_executionCount++;

#endif

// Don't allow loop headers to be released while the function is executing

::InterlockedIncrement(this->m\_depth.AddressOf());

}

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

///

/// FunctionBody::CheckEmpty

///

/// CheckEmpty() validates that the given instance has not been given an

/// implementation yet.

///

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

void

FunctionBody::CheckEmpty()

{

AssertMsg((this->byteCodeBlock == nullptr) && (this->auxBlock == nullptr) && (this->auxContextBlock == nullptr), "Function body may only be set once");

}

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

///

/// FunctionBody::CheckNotExecuting

///

/// CheckNotExecuting() checks that function is not currently executing when it

/// is being modified. See BeginExecution() for details.

///

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

void

FunctionBody::CheckNotExecuting()

{

AssertMsg(m\_DEBUG\_executionCount == 0, "Function cannot be executing when modified");

}

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

///

/// FunctionBody::EndExecution

///

/// EndExecution() is called by InterpreterStackFrame when a function ends execution.

/// See BeginExecution() for details.

///

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

void

FunctionBody::EndExecution()

{

#if DBG

AssertMsg(m\_DEBUG\_executionCount > 0, "Must have a previous execution to end");

m\_DEBUG\_executionCount--;

#endif

uint depth = ::InterlockedDecrement(this->m\_depth.AddressOf());

// If loop headers were determined to be no longer needed

// during the execution of the function, we release them now

if (depth == 0 && this->m\_pendingLoopHeaderRelease)

{

this->m\_pendingLoopHeaderRelease = false;

ReleaseLoopHeaders();

}

}

void FunctionBody::AddEntryPointToEntryPointList(FunctionEntryPointInfo\* entryPointInfo)

{

ThreadContext::AutoDisableExpiration disableExpiration(this->m\_scriptContext->GetThreadContext());

Recycler\* recycler = this->m\_scriptContext->GetRecycler();

entryPointInfo->entryPointIndex = this->entryPoints->Add(recycler->CreateWeakReferenceHandle(entryPointInfo));

}

BOOL FunctionBody::IsInterpreterThunk() const

{

bool isInterpreterThunk = this->originalEntryPoint == DefaultEntryThunk;

#if DYNAMIC\_INTERPRETER\_THUNK

isInterpreterThunk = isInterpreterThunk || IsDynamicInterpreterThunk();

#endif

return isInterpreterThunk;

}

BOOL FunctionBody::IsDynamicInterpreterThunk() const

{

#if DYNAMIC\_INTERPRETER\_THUNK

return this->GetScriptContext()->IsDynamicInterpreterThunk(this->originalEntryPoint);

#else

return FALSE;

#endif

}

FunctionEntryPointInfo \* FunctionBody::TryGetEntryPointInfo(int index) const

{

// If we've already freed the recyclable data, we're shutting down the script context so skip clean up

if (this->entryPoints == nullptr) return 0;

Assert(index < this->entryPoints->Count());

FunctionEntryPointInfo\* entryPoint = this->entryPoints->Item(index)->Get();

return entryPoint;

}

FunctionEntryPointInfo \* FunctionBody::GetEntryPointInfo(int index) const

{

FunctionEntryPointInfo\* entryPoint = TryGetEntryPointInfo(index);

Assert(entryPoint);

return entryPoint;

}

uint32 FunctionBody::GetFrameHeight(EntryPointInfo\* entryPointInfo) const

{

return entryPointInfo->frameHeight;

}

void FunctionBody::SetFrameHeight(EntryPointInfo\* entryPointInfo, uint32 frameHeight)

{

entryPointInfo->frameHeight = frameHeight;

}

#if ENABLE\_NATIVE\_CODEGEN

void

FunctionBody::SetNativeThrowSpanSequence(SmallSpanSequence \*seq, uint loopNum, LoopEntryPointInfo\* entryPoint)

{

Assert(loopNum != LoopHeader::NoLoop);

LoopHeader \*loopHeader = this->GetLoopHeader(loopNum);

Assert(loopHeader);

Assert(entryPoint->loopHeader == loopHeader);

entryPoint->SetNativeThrowSpanSequence(seq);

}

void

FunctionBody::RecordNativeThrowMap(SmallSpanSequenceIter& iter, uint32 nativeOffset, uint32 statementIndex, EntryPointInfo\* entryPoint, uint loopNum)

{

SmallSpanSequence \*pSpanSequence;

pSpanSequence = entryPoint->GetNativeThrowSpanSequence();

if (!pSpanSequence)

{

if (statementIndex == -1)

{

return; // No need to initialize native throw map for non-user code

}

pSpanSequence = HeapNew(SmallSpanSequence);

if (loopNum == LoopHeader::NoLoop)

{

((FunctionEntryPointInfo\*) entryPoint)->SetNativeThrowSpanSequence(pSpanSequence);

}

else

{

this->SetNativeThrowSpanSequence(pSpanSequence, loopNum, (LoopEntryPointInfo\*) entryPoint);

}

}

else if (iter.accumulatedSourceBegin == static\_cast<int>(statementIndex))

{

return; // Compress adjacent spans which share the same statementIndex

}

StatementData data;

data.sourceBegin = static\_cast<int>(statementIndex); // sourceBegin represents statementIndex here

data.bytecodeBegin = static\_cast<int>(nativeOffset); // bytecodeBegin represents nativeOffset here

pSpanSequence->RecordARange(iter, &data);

}

#endif

bool

ParseableFunctionInfo::IsTrackedPropertyId(PropertyId pid)

{

Assert(this->m\_boundPropertyRecords != nullptr);

PropertyRecordList\* trackedProperties = this->m\_boundPropertyRecords;

const PropertyRecord\* prop = nullptr;

if (trackedProperties->TryGetValue(pid, &prop))

{

Assert(prop != nullptr);

return true;

}

return this->m\_scriptContext->IsTrackedPropertyId(pid);

}

PropertyId

ParseableFunctionInfo::GetOrAddPropertyIdTracked(JsUtil::CharacterBuffer<WCHAR> const& propName)

{

Assert(this->m\_boundPropertyRecords != nullptr);

const Js::PropertyRecord\* propRecord = nullptr;

this->m\_scriptContext->GetOrAddPropertyRecord(propName, &propRecord);

PropertyId pid = propRecord->GetPropertyId();

this->m\_boundPropertyRecords->Item(pid, propRecord);

return pid;

}

#if ENABLE\_NATIVE\_CODEGEN

void

FunctionBody::RecordNativeBaseAddress(BYTE\* baseAddress, ptrdiff\_t size, NativeCodeData \* data, NativeCodeData \* transferData,

CodeGenNumberChunk \* numberChunks, EntryPointInfo\* entryPoint, uint loopNum)

{

entryPoint->SetCodeGenRecorded(baseAddress, size, data, transferData, numberChunks);

}

#endif

int

FunctionBody::GetNextDebuggerScopeIndex()

{

return this->debuggerScopeIndex++;

}

SmallSpanSequence::SmallSpanSequence()

: pStatementBuffer(nullptr),

pActualOffsetList(nullptr),

baseValue(0)

{

}

BOOL SmallSpanSequence::RecordARange(SmallSpanSequenceIter &iter, StatementData \* data)

{

Assert(data);

if (!this->pStatementBuffer)

{

this->pStatementBuffer = JsUtil::GrowingUint32HeapArray::Create(4);

baseValue = data->sourceBegin;

Reset(iter);

}

SmallSpan span(0);

span.sourceBegin = GetDiff(data->sourceBegin, iter.accumulatedSourceBegin);

span.bytecodeBegin = GetDiff(data->bytecodeBegin, iter.accumulatedBytecodeBegin);

this->pStatementBuffer->Add((uint32)span);

// Update iterator for the next set

iter.accumulatedSourceBegin = data->sourceBegin;

iter.accumulatedBytecodeBegin = data->bytecodeBegin;

return TRUE;

}

// FunctionProxy methods

ScriptContext\*

FunctionProxy::GetScriptContext() const

{

return m\_scriptContext;

}

void FunctionProxy::Copy(FunctionProxy\* other)

{

Assert(other);

other->SetIsTopLevel(this->m\_isTopLevel);

if (this->IsPublicLibraryCode())

{

other->SetIsPublicLibraryCode();

}

}

void ParseableFunctionInfo::Copy(FunctionBody\* other)

{

#define CopyDeferParseField(field) other->field = this->field;

CopyDeferParseField(m\_isDeclaration);

CopyDeferParseField(m\_isAccessor);

CopyDeferParseField(m\_isStrictMode);

CopyDeferParseField(m\_isGlobalFunc);

CopyDeferParseField(m\_doBackendArgumentsOptimization);

CopyDeferParseField(m\_isEval);

CopyDeferParseField(m\_isDynamicFunction);

CopyDeferParseField(m\_hasImplicitArgIns);

CopyDeferParseField(m\_dontInline);

CopyDeferParseField(m\_inParamCount);

CopyDeferParseField(m\_grfscr);

CopyDeferParseField(m\_scopeInfo);

CopyDeferParseField(m\_utf8SourceHasBeenSet);

#if DBG

CopyDeferParseField(deferredParseNextFunctionId);

CopyDeferParseField(scopeObjectSize);

#endif

CopyDeferParseField(scopeSlotArraySize);

CopyDeferParseField(cachedSourceString);

CopyDeferParseField(deferredStubs);

CopyDeferParseField(m\_isAsmjsMode);

CopyDeferParseField(m\_isAsmJsFunction);

#undef CopyDeferParseField

other->CopySourceInfo(this);

}

void FunctionProxy::SetReferenceInParentFunction(FunctionProxyPtrPtr reference)

{

if (reference)

{

// Tag the reference so that the child function doesn't

// keep the parent alive. If the parent function is going away,

// it'll clear its children's references.

this->m\_referenceInParentFunction = reference;

}

else

{

this->m\_referenceInParentFunction = nullptr;

}

}

void FunctionProxy::UpdateReferenceInParentFunction(FunctionProxy\* newFunctionInfo)

{

if (this->m\_referenceInParentFunction)

{

#ifdef RECYCLER\_WRITE\_BARRIER

if (newFunctionInfo == nullptr)

{

(\*m\_referenceInParentFunction).NoWriteBarrierSet(nullptr);

return;

}

#endif

(\*m\_referenceInParentFunction) = newFunctionInfo;

}

}

// DeferDeserializeFunctionInfo methods

DeferDeserializeFunctionInfo::DeferDeserializeFunctionInfo(int nestedCount, LocalFunctionId functionId, ByteCodeCache\* byteCodeCache, const byte\* serializedFunction, Utf8SourceInfo\* sourceInfo, ScriptContext\* scriptContext, uint functionNumber, const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, NativeModule \*nativeModule, Attributes attributes) :

FunctionProxy(DefaultDeferredDeserializeThunk, (Attributes)(attributes | DeferredDeserialize), nestedCount, sizeof(DeferDeserializeFunctionInfo), functionId, scriptContext, sourceInfo, functionNumber),

m\_cache(byteCodeCache),

m\_functionBytes(serializedFunction),

m\_displayName(nullptr),

m\_displayNameLength(0),

m\_nativeModule(nativeModule)

{

this->m\_defaultEntryPointInfo = RecyclerNew(scriptContext->GetRecycler(), ProxyEntryPointInfo, DefaultDeferredDeserializeThunk);

PERF\_COUNTER\_INC(Code, DeferDeserializeFunctionProxy);

SetDisplayName(displayName, displayNameLength, displayShortNameOffset, FunctionProxy::SetDisplayNameFlagsDontCopy);

}

DeferDeserializeFunctionInfo\* DeferDeserializeFunctionInfo::New(ScriptContext\* scriptContext, int nestedCount, LocalFunctionId functionId, ByteCodeCache\* byteCodeCache, const byte\* serializedFunction, Utf8SourceInfo\* sourceInfo, const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, NativeModule \*nativeModule, Attributes attributes)

{

return RecyclerNewFinalized(scriptContext->GetRecycler(),

DeferDeserializeFunctionInfo,

nestedCount,

functionId,

byteCodeCache,

serializedFunction,

sourceInfo,

scriptContext,

scriptContext->GetThreadContext()->NewFunctionNumber(),

displayName,

displayNameLength,

displayShortNameOffset,

nativeModule,

attributes);

}

const wchar\_t\*

DeferDeserializeFunctionInfo::GetDisplayName() const

{

return this->m\_displayName;

}

// ParseableFunctionInfo methods

ParseableFunctionInfo::ParseableFunctionInfo(JavascriptMethod entryPoint, int nestedCount, int derivedSize,

LocalFunctionId functionId, Utf8SourceInfo\* sourceInfo, ScriptContext\* scriptContext, uint functionNumber,

const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, Attributes attributes, Js::PropertyRecordList\* propertyRecords) :

FunctionProxy(entryPoint, attributes, nestedCount, derivedSize, functionId, scriptContext, sourceInfo, functionNumber),

#if DYNAMIC\_INTERPRETER\_THUNK

m\_dynamicInterpreterThunk(nullptr),

#endif

m\_hasBeenParsed(false),

m\_isGlobalFunc(false),

m\_isDeclaration(false),

m\_isNamedFunctionExpression(false),

m\_isNameIdentifierRef (true),

m\_isStaticNameFunction(false),

m\_doBackendArgumentsOptimization(true),

m\_isStrictMode(false),

m\_isAsmjsMode(false),

m\_dontInline(false),

m\_hasImplicitArgIns(true),

m\_grfscr(0),

m\_inParamCount(0),

m\_reportedInParamCount(0),

m\_sourceIndex(Js::Constants::InvalidSourceIndex),

m\_utf8SourceHasBeenSet(false),

m\_cchLength(0),

m\_cbLength(0),

m\_cchStartOffset(0),

m\_cbStartOffset(0),

m\_lineNumber(0),

m\_columnNumber(0),

m\_isEval(false),

m\_isDynamicFunction(false),

m\_scopeInfo(nullptr),

m\_displayName(nullptr),

m\_displayNameLength(0),

m\_displayShortNameOffset(0),

deferredStubs(nullptr),

scopeSlotArraySize(0),

cachedSourceString(nullptr),

m\_boundPropertyRecords(propertyRecords),

m\_reparsed(false),

m\_isAsmJsFunction(false),

#if DBG

m\_wasEverAsmjsMode(false),

scopeObjectSize(0),

#endif

isByteCodeDebugMode(false)

{

if ((attributes & Js::FunctionInfo::DeferredParse) == 0)

{

void\* validationCookie = nullptr;

#if ENABLE\_NATIVE\_CODEGEN

validationCookie = (void\*)scriptContext->GetNativeCodeGenerator();

#endif

this->m\_defaultEntryPointInfo = RecyclerNewFinalized(scriptContext->GetRecycler(),

FunctionEntryPointInfo, this, entryPoint, scriptContext->GetThreadContext(), validationCookie);

}

else

{

this->m\_defaultEntryPointInfo = RecyclerNew(scriptContext->GetRecycler(), ProxyEntryPointInfo, entryPoint);

}

SetDisplayName(displayName, displayNameLength, displayShortNameOffset);

this->originalEntryPoint = DefaultEntryThunk;

}

ParseableFunctionInfo\* ParseableFunctionInfo::New(ScriptContext\* scriptContext, int nestedCount,

LocalFunctionId functionId, Utf8SourceInfo\* sourceInfo, const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, Js::PropertyRecordList\* propertyRecords, Attributes attributes)

{

Assert(scriptContext->DeferredParsingThunk == ProfileDeferredParsingThunk

|| scriptContext->DeferredParsingThunk == DefaultDeferredParsingThunk);

#ifdef PERF\_COUNTERS

PERF\_COUNTER\_INC(Code, DeferedFunction);

#endif

uint newFunctionNumber = scriptContext->GetThreadContext()->NewFunctionNumber();

if (!sourceInfo->GetSourceContextInfo()->IsDynamic())

{

PHASE\_PRINT\_TESTTRACE1(Js::DeferParsePhase, L"Function was deferred from parsing - ID: %d; Display Name: %s; Utf8SourceInfo ID: %d; Source Length: %d; Source Url:%s\n", newFunctionNumber, displayName, sourceInfo->GetSourceInfoId(), sourceInfo->GetCchLength(), sourceInfo->GetSourceContextInfo()->url);

}

else

{

PHASE\_PRINT\_TESTTRACE1(Js::DeferParsePhase, L"Function was deferred from parsing - ID: %d; Display Name: %s; Utf8SourceInfo ID: %d; Source Length: %d;\n", newFunctionNumber, displayName, sourceInfo->GetSourceInfoId(), sourceInfo->GetCchLength());

}

// When generating a new defer parse function, we always use a new function number

return RecyclerNewWithBarrierFinalizedPlus(scriptContext->GetRecycler(),

nestedCount \* sizeof(FunctionBody\*),

ParseableFunctionInfo,

scriptContext->DeferredParsingThunk,

nestedCount,

sizeof(ParseableFunctionInfo),

functionId,

sourceInfo,

scriptContext,

newFunctionNumber,

displayName,

displayNameLength,

displayShortNameOffset,

(Attributes)(attributes | DeferredParse),

propertyRecords);

}

DWORD\_PTR FunctionProxy::GetSecondaryHostSourceContext() const

{

return this->m\_utf8SourceInfo->GetSecondaryHostSourceContext();

}

DWORD\_PTR FunctionProxy::GetHostSourceContext() const

{

return this->GetSourceContextInfo()->dwHostSourceContext;

}

SourceContextInfo \* FunctionProxy::GetSourceContextInfo() const

{

return this->GetHostSrcInfo()->sourceContextInfo;

}

SRCINFO const \* FunctionProxy::GetHostSrcInfo() const

{

return m\_utf8SourceInfo->GetSrcInfo();

}

//

// Returns the start line for the script buffer (code buffer for the entire script tag) of this current function.

// We subtract the lnMinHost because it is the number of lines we have added to augment scriptlets passed through

// ParseProcedureText to have a function name.

//

ULONG FunctionProxy::GetHostStartLine() const

{

return this->GetHostSrcInfo()->dlnHost - this->GetHostSrcInfo()->lnMinHost;

}

//

// Returns the start column of the first line for the script buffer of this current function.

//

ULONG FunctionProxy::GetHostStartColumn() const

{

return this->GetHostSrcInfo()->ulColumnHost;

}

//

// Returns line number in unmodified host buffer (i.e. without extra scriptlet code added by ParseProcedureText --

// when e.g. we add extra code for event handlers, such as "function onclick(event)\n{\n").

//

ULONG FunctionProxy::GetLineNumberInHostBuffer(ULONG relativeLineNumber) const

{

ULONG lineNumber = relativeLineNumber;

if (lineNumber >= this->GetHostSrcInfo()->lnMinHost)

{

lineNumber -= this->GetHostSrcInfo()->lnMinHost;

}

// Note that '<' is still a valid case -- that would be the case for onclick scriptlet function itself (lineNumber == 0).

return lineNumber;

}

ULONG FunctionProxy::ComputeAbsoluteLineNumber(ULONG relativeLineNumber) const

{

// We add 1 because the line numbers start from 0.

return this->GetHostSrcInfo()->dlnHost + GetLineNumberInHostBuffer(relativeLineNumber) + 1;

}

ULONG FunctionProxy::ComputeAbsoluteColumnNumber(ULONG relativeLineNumber, ULONG relativeColumnNumber) const

{

if (this->GetLineNumberInHostBuffer(relativeLineNumber) == 0)

{

// The host column matters only for the first line.

return this->GetHostStartColumn() + relativeColumnNumber + 1;

}

// We add 1 because the column numbers start from 0.

return relativeColumnNumber + 1;

}

//

// Returns the line number of the function declaration in the source file.

//

ULONG

ParseableFunctionInfo::GetLineNumber() const

{

return this->ComputeAbsoluteLineNumber(this->m\_lineNumber);

}

//

// Returns the column number of the function declaration in the source file.

//

ULONG

ParseableFunctionInfo::GetColumnNumber() const

{

return ComputeAbsoluteColumnNumber(this->m\_lineNumber, m\_columnNumber);

}

LPCWSTR

ParseableFunctionInfo::GetSourceName() const

{

return GetSourceName(this->GetSourceContextInfo());

}

void

ParseableFunctionInfo::SetGrfscr(ulong grfscr)

{

this->m\_grfscr = grfscr;

}

ulong

ParseableFunctionInfo::GetGrfscr() const

{

return this->m\_grfscr;

}

ProxyEntryPointInfo\*

FunctionProxy::GetDefaultEntryPointInfo() const

{

return this->m\_defaultEntryPointInfo;

}

FunctionEntryPointInfo\*

FunctionBody::GetDefaultFunctionEntryPointInfo() const

{

Assert(((ProxyEntryPointInfo\*) this->defaultFunctionEntryPointInfo) == this->m\_defaultEntryPointInfo);

return this->defaultFunctionEntryPointInfo;

}

void

ParseableFunctionInfo::SetInParamsCount(ArgSlot newInParamCount)

{

AssertMsg(m\_inParamCount <= newInParamCount, "Cannot shrink register usage");

m\_inParamCount = newInParamCount;

if (newInParamCount <= 1)

{

SetHasImplicitArgIns(false);

}

}

ArgSlot

ParseableFunctionInfo::GetReportedInParamsCount() const

{

return m\_reportedInParamCount;

}

void

ParseableFunctionInfo::SetReportedInParamsCount(ArgSlot newInParamCount)

{

AssertMsg(m\_reportedInParamCount <= newInParamCount, "Cannot shrink register usage");

m\_reportedInParamCount = newInParamCount;

}

void

ParseableFunctionInfo::ResetInParams()

{

m\_inParamCount = 0;

m\_reportedInParamCount = 0;

}

const wchar\_t\*

ParseableFunctionInfo::GetDisplayName() const

{

return this->m\_displayName;

}

void ParseableFunctionInfo::BuildDeferredStubs(ParseNode \*pnodeFnc)

{

Assert(pnodeFnc->nop == knopFncDecl);

Recycler \*recycler = GetScriptContext()->GetRecycler();

this->deferredStubs = BuildDeferredStubTree(pnodeFnc, recycler);

}

FunctionProxyArray ParseableFunctionInfo::GetNestedFuncArray()

{

// The array is allocated as extra bytes past the end of the struct.

Assert(this->m\_nestedCount > 0);

return (FunctionProxyArray )((char\*)this + m\_derivedSize);

}

void ParseableFunctionInfo::SetNestedFunc(FunctionProxy\* nestedFunc, uint index, ulong flags)

{

AssertMsg(index < this->m\_nestedCount, "Trying to write past the nested func array");

FunctionProxyArray nested = this->GetNestedFuncArray();

nested[index] = nestedFunc;

if (nestedFunc)

{

nestedFunc->SetReferenceInParentFunction(GetNestedFuncReference(index));

if (!this->GetSourceContextInfo()->IsDynamic() && nestedFunc->IsDeferredParseFunction() && nestedFunc->GetParseableFunctionInfo()->GetIsDeclaration() && this->GetIsTopLevel() && !(flags & fscrEvalCode))

{

this->m\_utf8SourceInfo->TrackDeferredFunction(nestedFunc->GetLocalFunctionId(), nestedFunc->GetParseableFunctionInfo());

}

}

}

FunctionProxy\* ParseableFunctionInfo::GetNestedFunc(uint index)

{

return \*(GetNestedFuncReference(index));

}

FunctionProxyPtrPtr ParseableFunctionInfo::GetNestedFuncReference(uint index)

{

AssertMsg(index < this->m\_nestedCount, "Trying to write past the nested func array");

FunctionProxyArray nested = this->GetNestedFuncArray();

return &nested[index];

}

ParseableFunctionInfo\* ParseableFunctionInfo::GetNestedFunctionForExecution(uint index)

{

FunctionProxy\* currentNestedFunction = this->GetNestedFunc(index);

Assert(currentNestedFunction);

if (currentNestedFunction->IsDeferredDeserializeFunction())

{

currentNestedFunction = currentNestedFunction->EnsureDeserialized();

this->SetNestedFunc(currentNestedFunction, index, 0u);

}

return currentNestedFunction->GetParseableFunctionInfo();

}

void

FunctionProxy::UpdateFunctionBodyImpl(FunctionBody \* body)

{

Assert(functionBodyImpl == ((FunctionProxy\*) this));

Assert(!this->IsFunctionBody() || body == this);

this->functionBodyImpl = body;

this->attributes = (Attributes)(this->attributes & ~(DeferredParse | DeferredDeserialize));

this->UpdateReferenceInParentFunction(body);

}

void ParseableFunctionInfo::ClearNestedFunctionParentFunctionReference()

{

if (this->m\_nestedCount > 0)

{

// If the function is x-domain all the nested functions should also be marked as x-domain

FunctionProxyArray nested = this->GetNestedFuncArray();

for (uint i = 0; i < this->m\_nestedCount; ++i)

{

if (nested[i])

{

nested[i]->SetReferenceInParentFunction(nullptr);

}

}

}

}

//

// This method gets a function body for the purposes of execution

// It has an if within it to avoid making it a virtual- it's called from the interpreter

// It will cause the function info to get deserialized if it hasn't been deserialized

// already

//

ParseableFunctionInfo \* FunctionProxy::EnsureDeserialized()

{

FunctionProxy \* executionFunctionBody = this->functionBodyImpl;

if (executionFunctionBody == this && IsDeferredDeserializeFunction())

{

// No need to deserialize function body if scriptContext closed because we can't execute it.

// Bigger problem is the script engine might have released bytecode file mapping and we can't deserialize.

Assert(!m\_scriptContext->IsClosed());

executionFunctionBody = ((DeferDeserializeFunctionInfo\*) this)->Deserialize();

this->functionBodyImpl = executionFunctionBody;

Assert(executionFunctionBody->HasBody());

Assert(executionFunctionBody != this);

}

return (ParseableFunctionInfo \*)executionFunctionBody;

}

ScriptFunctionType \* FunctionProxy::GetDeferredPrototypeType() const

{

return deferredPrototypeType;

}

ScriptFunctionType \* FunctionProxy::EnsureDeferredPrototypeType()

{

Assert(this->GetFunctionProxy() == this);

return (deferredPrototypeType != nullptr)? deferredPrototypeType : AllocDeferredPrototypeType();

}

ScriptFunctionType \* FunctionProxy::AllocDeferredPrototypeType()

{

Assert(deferredPrototypeType == nullptr);

ScriptFunctionType \* type = ScriptFunctionType::New(this, true);

deferredPrototypeType = type;

return type;

}

JavascriptMethod FunctionProxy::GetDirectEntryPoint(ProxyEntryPointInfo\* entryPoint) const

{

Assert((JavascriptMethod)entryPoint->address != nullptr);

return (JavascriptMethod)entryPoint->address;

}

// Function object type list methods

template <typename Fn>

void FunctionProxy::MapFunctionObjectTypes(Fn func)

{

if (m\_functionObjectTypeList)

{

m\_functionObjectTypeList->Map([&] (int, FunctionTypeWeakRef\* typeWeakRef)

{

if (typeWeakRef)

{

DynamicType\* type = typeWeakRef->Get();

if (type)

{

func(type);

}

}

});

}

if (this->deferredPrototypeType)

{

func(this->deferredPrototypeType);

}

}

FunctionProxy::FunctionTypeWeakRefList\* FunctionProxy::EnsureFunctionObjectTypeList()

{

if (m\_functionObjectTypeList == nullptr)

{

Recycler\* recycler = this->GetScriptContext()->GetRecycler();

m\_functionObjectTypeList = RecyclerNew(recycler, FunctionTypeWeakRefList, recycler);

}

return m\_functionObjectTypeList;

}

void FunctionProxy::RegisterFunctionObjectType(DynamicType\* functionType)

{

FunctionTypeWeakRefList\* typeList = EnsureFunctionObjectTypeList();

Assert(functionType != deferredPrototypeType);

Recycler \* recycler = this->GetScriptContext()->GetRecycler();

FunctionTypeWeakRef\* weakRef = recycler->CreateWeakReferenceHandle(functionType);

typeList->SetAtFirstFreeSpot(weakRef);

OUTPUT\_TRACE(Js::ExpirableCollectPhase, L"Registered type 0x%p on function body %p, count = %d\n", functionType, this, typeList->Count());

}

void DeferDeserializeFunctionInfo::SetDisplayName(const wchar\_t\* displayName)

{

size\_t len = wcslen(displayName);

if (len > UINT\_MAX)

{

// Can't support display name that big

Js::Throw::OutOfMemory();

}

SetDisplayName(displayName, (uint)len, 0);

}

void DeferDeserializeFunctionInfo::SetDisplayName(const wchar\_t\* pszDisplayName, uint displayNameLength, uint displayShortNameOffset, SetDisplayNameFlags flags /\* default to None \*/)

{

this->m\_displayNameLength = displayNameLength;

this->m\_displayShortNameOffset = displayShortNameOffset;

FunctionProxy::SetDisplayName(pszDisplayName, &this->m\_displayName, displayNameLength, m\_scriptContext, flags);

}

LPCWSTR DeferDeserializeFunctionInfo::GetSourceInfo(int& lineNumber, int& columnNumber) const

{

// Read all the necessary information from the serialized byte code

int lineNumberField, columnNumberField;

bool m\_isEval, m\_isDynamicFunction;

ByteCodeSerializer::ReadSourceInfo(this, lineNumberField, columnNumberField, m\_isEval, m\_isDynamicFunction);

// Decode them

lineNumber = ComputeAbsoluteLineNumber(lineNumberField);

columnNumber = ComputeAbsoluteColumnNumber(lineNumberField, columnNumberField);

return Js::ParseableFunctionInfo::GetSourceName<SourceContextInfo\*>(this->GetSourceContextInfo(), m\_isEval, m\_isDynamicFunction);

}

void DeferDeserializeFunctionInfo::Finalize(bool isShutdown)

{

\_\_super::Finalize(isShutdown);

PERF\_COUNTER\_DEC(Code, DeferDeserializeFunctionProxy);

}

FunctionBody\* DeferDeserializeFunctionInfo::Deserialize()

{

if (functionBodyImpl == (FunctionBody\*) this)

{

FunctionBody \* body = ByteCodeSerializer::DeserializeFunction(this->m\_scriptContext, this);

this->Copy(body);

this->UpdateFunctionBodyImpl(body);

}

return GetFunctionBody();

}

//

// hrParse can be one of the following from deferred re-parse (check CompileScriptException::ProcessError):

// E\_OUTOFMEMORY

// E\_UNEXPECTED

// SCRIPT\_E\_RECORDED,

// with ei.scode: ERRnoMemory, VBSERR\_OutOfStack, E\_OUTOFMEMORY, E\_FAIL

// Any other ei.scode shouldn't appear in deferred re-parse.

//

// Map errors like OOM/SOE, return it and clean hrParse. Any other error remaining in hrParse is an internal error.

//

HRESULT ParseableFunctionInfo::MapDeferredReparseError(HRESULT& hrParse, const CompileScriptException& se)

{

HRESULT hrMapped = NO\_ERROR;

switch (hrParse)

{

case E\_OUTOFMEMORY:

hrMapped = E\_OUTOFMEMORY;

break;

case SCRIPT\_E\_RECORDED:

switch (se.ei.scode)

{

case ERRnoMemory:

case E\_OUTOFMEMORY:

case VBSERR\_OutOfMemory:

hrMapped = E\_OUTOFMEMORY;

break;

case VBSERR\_OutOfStack:

hrMapped = VBSERR\_OutOfStack;

break;

}

}

if (FAILED(hrMapped))

{

// If we have mapped error, clear hrParse. We'll throw error from hrMapped.

hrParse = NO\_ERROR;

}

return hrMapped;

}

FunctionBody\* ParseableFunctionInfo::Parse(ScriptFunction \*\* functionRef, bool isByteCodeDeserialization)

{

if ((functionBodyImpl != (FunctionBody\*) this) || !IsDeferredParseFunction())

{

// If not deferredparsed, the functionBodyImpl and this will be the same, just return the current functionBody.

Assert(GetFunctionBody()->IsFunctionParsed());

return GetFunctionBody();

}

BOOL fParsed = FALSE;

FunctionBody\* returnFunctionBody = nullptr;

ENTER\_PINNED\_SCOPE(Js::PropertyRecordList, propertyRecordList);

Recycler\* recycler = this->m\_scriptContext->GetRecycler();

propertyRecordList = RecyclerNew(recycler, Js::PropertyRecordList, recycler);

bool isDebugReparse = m\_scriptContext->IsInDebugOrSourceRundownMode() && !this->m\_utf8SourceInfo->GetIsLibraryCode();

bool isAsmJsReparse = false;

bool isReparse = isDebugReparse;

FunctionBody\* funcBody = nullptr;

// If m\_hasBeenParsed = true, one of the following things happened things happened:

// - We had multiple function objects which were all defer-parsed, but with the same function body and one of them

// got the body to be parsed before another was called

// - We are in debug mode and had our thunks switched to DeferParseThunk

// - This is an already parsed asm.js module, which has been invalidated at link time and must be reparsed as a non-asm.js function

if (!this->m\_hasBeenParsed)

{

funcBody = FunctionBody::NewFromRecycler(

this->m\_scriptContext,

this->m\_displayName,

this->m\_displayNameLength,

this->m\_displayShortNameOffset,

this->m\_nestedCount,

this->m\_utf8SourceInfo,

this->m\_functionNumber,

m\_utf8SourceInfo->GetSrcInfo()->sourceContextInfo->sourceContextId, /\* script id \*/

this->functionId, /\* function id \*/

propertyRecordList,

(Attributes)(this->GetAttributes() & ~(Attributes::DeferredDeserialize | Attributes::DeferredParse))

#ifdef PERF\_COUNTERS

, false /\* is function from deferred deserialized proxy \*/

#endif

);

this->Copy(funcBody);

PERF\_COUNTER\_DEC(Code, DeferedFunction);

if (!this->GetSourceContextInfo()->IsDynamic())

{

PHASE\_PRINT\_TESTTRACE1(Js::DeferParsePhase, L"TestTrace: Deferred function parsed - ID: %d; Display Name: %s; Length: %d; Nested Function Count: %d; Utf8SourceInfo: %d; Source Length: %d; Is Top Level: %s; Source Url: %s\n", m\_functionNumber, m\_displayName, this->m\_cchLength, this->GetNestedCount(), this->m\_utf8SourceInfo->GetSourceInfoId(), this->m\_utf8SourceInfo->GetCchLength(), this->GetIsTopLevel() ? L"True" : L"False", this->GetSourceContextInfo()->url);

}

else

{

PHASE\_PRINT\_TESTTRACE1(Js::DeferParsePhase, L"TestTrace: Deferred function parsed - ID: %d; Display Name: %s; Length: %d; Nested Function Count: %d; Utf8SourceInfo: %d; Source Length: %d\n; Is Top Level: %s;", m\_functionNumber, m\_displayName, this->m\_cchLength, this->GetNestedCount(), this->m\_utf8SourceInfo->GetSourceInfoId(), this->m\_utf8SourceInfo->GetCchLength(), this->GetIsTopLevel() ? L"True" : L"False");

}

if (!this->GetIsTopLevel() &&

!this->GetSourceContextInfo()->IsDynamic() &&

this->m\_scriptContext->DoUndeferGlobalFunctions())

{

this->m\_utf8SourceInfo->UndeferGlobalFunctions([this](JsUtil::SimpleDictionaryEntry<Js::LocalFunctionId, Js::ParseableFunctionInfo\*> func)

{

Js::ParseableFunctionInfo \*nextFunc = func.Value();

JavascriptExceptionObject\* pExceptionObject = nullptr;

if (nextFunc != nullptr && this != nextFunc)

{

try

{

nextFunc->Parse();

}

catch (OutOfMemoryException) {}

catch (StackOverflowException) {}

catch (JavascriptExceptionObject\* exceptionObject)

{

pExceptionObject = exceptionObject;

}

// Do not do anything with an OOM or SOE, returning true is fine, it will then be undeferred (or attempted to again when called)

if(pExceptionObject)

{

if(pExceptionObject != ThreadContext::GetContextForCurrentThread()->GetPendingOOMErrorObject() &&

pExceptionObject != ThreadContext::GetContextForCurrentThread()->GetPendingSOErrorObject())

{

throw pExceptionObject;

}

}

}

return true;

});

}

}

else

{

isAsmJsReparse = m\_isAsmjsMode && !isDebugReparse;

isReparse |= isAsmJsReparse;

funcBody = this->GetFunctionBody();

if (isReparse)

{

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

#endif

#if DBG

Assert(

funcBody->IsReparsed()

|| m\_scriptContext->IsInDebugOrSourceRundownMode()

|| m\_isAsmjsMode);

#endif

OUTPUT\_TRACE(Js::DebuggerPhase, L"Full nested reparse of function: %s (%s)\n", funcBody->GetDisplayName(), funcBody->GetDebugNumberSet(debugStringBuffer));

if (funcBody->GetByteCode())

{

// The current function needs to be cleaned up before getting generated in the debug mode.

funcBody->CleanupToReparse();

}

}

}

// Note that we may be trying to re-gen an already-completed function. (This can happen, for instance,

// in the case of named function expressions inside "with" statements in compat mode.)

// In such a case, there's no work to do.

if (funcBody->GetByteCode() == nullptr)

{

#if ENABLE\_PROFILE\_INFO

Assert(!funcBody->HasExecutionDynamicProfileInfo());

#endif

// In debug mode, the eval code will be asked to recompile again.

AssertMsg(isDebugReparse || !(funcBody->GetGrfscr() & (fscrImplicitThis | fscrImplicitParents)),

"Deferred parsing of event handler body?");

// In debug or asm.js mode, the scriptlet will be asked to recompile again.

AssertMsg(isReparse || funcBody->GetGrfscr() & fscrGlobalCode || CONFIG\_FLAG(DeferNested), "Deferred parsing of non-global procedure?");

HRESULT hr = NO\_ERROR;

HRESULT hrParser = NO\_ERROR;

HRESULT hrParseCodeGen = NO\_ERROR;

BEGIN\_LEAVE\_SCRIPT\_INTERNAL(m\_scriptContext)

{

bool isCesu8 = m\_scriptContext->GetSource(funcBody->GetSourceIndex())->IsCesu8();

size\_t offset = this->StartOffset();

charcount\_t charOffset = this->StartInDocument();

size\_t length = this->LengthInBytes();

LPCUTF8 pszStart = this->GetStartOfDocument();

ulong grfscr = funcBody->GetGrfscr() | fscrDeferredFnc;

// For the global function we want to re-use the glo functionbody which is already created in the non-debug mode

if (!funcBody->GetIsGlobalFunc())

{

grfscr &= ~fscrGlobalCode;

}

if (!funcBody->GetIsDeclaration() && !funcBody->GetIsGlobalFunc()) // No refresh may reparse global function (e.g. eval code)

{

// Notify the parser that the top-level function was defined in an expression,

// (not a function declaration statement).

grfscr |= fscrDeferredFncExpression;

}

if (!CONFIG\_FLAG(DeferNested) || isDebugReparse || isAsmJsReparse)

{

grfscr &= ~fscrDeferFncParse; // Disable deferred parsing if not DeferNested, or doing a debug/asm.js re-parse

}

if (isReparse)

{

grfscr |= fscrNoAsmJs; // Disable asm.js when debugging or if linking failed

}

BEGIN\_TRANSLATE\_EXCEPTION\_TO\_HRESULT

{

CompileScriptException se;

Parser ps(m\_scriptContext, funcBody->GetIsStrictMode() ? TRUE : FALSE);

ParseNodePtr parseTree;

uint nextFunctionId = funcBody->GetLocalFunctionId();

hrParser = ps.ParseSourceWithOffset(&parseTree, pszStart, offset, length, charOffset, isCesu8, grfscr, &se,

&nextFunctionId, funcBody->GetRelativeLineNumber(), funcBody->GetSourceContextInfo(),

funcBody, isReparse);

Assert(FAILED(hrParser) || nextFunctionId == funcBody->deferredParseNextFunctionId || isReparse || isByteCodeDeserialization);

if (FAILED(hrParser))

{

hrParseCodeGen = MapDeferredReparseError(hrParser, se); // Map certain errors like OOM/SOE

AssertMsg(FAILED(hrParseCodeGen) && SUCCEEDED(hrParser), "Syntax errors should never be detected on deferred re-parse");

}

else

{

TRACE\_BYTECODE(L"\nDeferred parse %s\n", funcBody->GetDisplayName());

Js::AutoDynamicCodeReference dynamicFunctionReference(m\_scriptContext);

bool forceNoNative = isReparse ? this->GetScriptContext()->IsInterpreted() : false;

hrParseCodeGen = GenerateByteCode(parseTree, grfscr, m\_scriptContext,

funcBody->GetParseableFunctionInfoRef(), funcBody->GetSourceIndex(),

forceNoNative, &ps, &se, funcBody->GetScopeInfo(), functionRef);

if (se.ei.scode == JSERR\_AsmJsCompileError)

{

// if asm.js compilation failed, reparse without asm.js

m\_grfscr |= fscrNoAsmJs;

se.Clear();

return Parse(functionRef, isByteCodeDeserialization);

}

if (SUCCEEDED(hrParseCodeGen))

{

fParsed = TRUE;

}

else

{

Assert(hrParseCodeGen == SCRIPT\_E\_RECORDED);

hrParseCodeGen = se.ei.scode;

}

}

}

END\_TRANSLATE\_EXCEPTION\_TO\_HRESULT(hr);

}

END\_LEAVE\_SCRIPT\_INTERNAL(m\_scriptContext);

if (hr == E\_OUTOFMEMORY)

{

JavascriptError::ThrowOutOfMemoryError(m\_scriptContext);

}

else if(hr == VBSERR\_OutOfStack)

{

JavascriptError::ThrowStackOverflowError(m\_scriptContext);

}

else if(hr == E\_ABORT)

{

throw Js::ScriptAbortException();

}

else if(FAILED(hr))

{

throw Js::InternalErrorException();

}

Assert(hr == NO\_ERROR);

if (!SUCCEEDED(hrParser))

{

JavascriptError::ThrowError(m\_scriptContext, VBSERR\_InternalError);

}

else if (!SUCCEEDED(hrParseCodeGen))

{

/\*

\* VBSERR\_OutOfStack is of type kjstError but we throw a (more specific) StackOverflowError when a hard stack

\* overflow occurs. To keep the behavior consistent I'm special casing it here.

\*/

if (hrParseCodeGen == VBSERR\_OutOfStack)

{

JavascriptError::ThrowStackOverflowError(m\_scriptContext);

}

JavascriptError::MapAndThrowError(m\_scriptContext, hrParseCodeGen);

}

}

else

{

fParsed = FALSE;

}

if (fParsed == TRUE)

{

this->UpdateFunctionBodyImpl(funcBody);

this->m\_hasBeenParsed = true;

}

returnFunctionBody = GetFunctionBody();

LEAVE\_PINNED\_SCOPE();

return returnFunctionBody;

}

#ifndef TEMP\_DISABLE\_ASMJS

FunctionBody\* ParseableFunctionInfo::ParseAsmJs(Parser \* ps, \_\_out CompileScriptException \* se, \_\_out ParseNodePtr \* parseTree)

{

Assert(IsDeferredParseFunction());

Assert(m\_isAsmjsMode);

FunctionBody\* returnFunctionBody = nullptr;

ENTER\_PINNED\_SCOPE(Js::PropertyRecordList, propertyRecordList);

Recycler\* recycler = this->m\_scriptContext->GetRecycler();

propertyRecordList = RecyclerNew(recycler, Js::PropertyRecordList, recycler);

FunctionBody\* funcBody = nullptr;

funcBody = FunctionBody::NewFromRecycler(

this->m\_scriptContext,

this->m\_displayName,

this->m\_displayNameLength,

this->m\_displayShortNameOffset,

this->m\_nestedCount,

this->m\_utf8SourceInfo,

this->m\_functionNumber,

m\_utf8SourceInfo->GetSrcInfo()->sourceContextInfo->sourceContextId, /\* script id \*/

this->functionId, /\* function id \*/

propertyRecordList,

(Attributes)(this->GetAttributes() & ~(Attributes::DeferredDeserialize | Attributes::DeferredParse))

#ifdef PERF\_COUNTERS

, false /\* is function from deferred deserialized proxy \*/

#endif

);

this->Copy(funcBody);

PERF\_COUNTER\_DEC(Code, DeferedFunction);

if (!this->GetSourceContextInfo()->IsDynamic())

{

PHASE\_PRINT\_TESTTRACE1(Js::DeferParsePhase, L"TestTrace: Deferred function parsed - ID: %d; Display Name: %s; Length: %d; Nested Function Count: %d; Utf8SourceInfo: %d; Source Length: %d; Is Top Level: %s; Source Url: %s\n", m\_functionNumber, m\_displayName, this->m\_cchLength, this->GetNestedCount(), this->m\_utf8SourceInfo->GetSourceInfoId(), this->m\_utf8SourceInfo->GetCchLength(), this->GetIsTopLevel() ? L"True" : L"False", this->GetSourceContextInfo()->url);

}

else

{

PHASE\_PRINT\_TESTTRACE1(Js::DeferParsePhase, L"TestTrace: Deferred function parsed - ID: %d; Display Name: %s; Length: %d; Nested Function Count: %d; Utf8SourceInfo: %d; Source Length: %d\n; Is Top Level: %s;", m\_functionNumber, m\_displayName, this->m\_cchLength, this->GetNestedCount(), this->m\_utf8SourceInfo->GetSourceInfoId(), this->m\_utf8SourceInfo->GetCchLength(), this->GetIsTopLevel() ? L"True" : L"False");

}

#if ENABLE\_PROFILE\_INFO

Assert(!funcBody->HasExecutionDynamicProfileInfo());

#endif

HRESULT hrParser = NO\_ERROR;

HRESULT hrParseCodeGen = NO\_ERROR;

bool isCesu8 = m\_scriptContext->GetSource(funcBody->GetSourceIndex())->IsCesu8();

size\_t offset = this->StartOffset();

charcount\_t charOffset = this->StartInDocument();

size\_t length = this->LengthInBytes();

LPCUTF8 pszStart = this->GetStartOfDocument();

ulong grfscr = funcBody->GetGrfscr() | fscrDeferredFnc | fscrDeferredFncExpression;

uint nextFunctionId = funcBody->GetLocalFunctionId();

// if parser throws, it will be caught by function trying to bytecode gen the asm.js module, so don't need to catch/rethrow here

hrParser = ps->ParseSourceWithOffset(parseTree, pszStart, offset, length, charOffset, isCesu8, grfscr, se,

&nextFunctionId, funcBody->GetRelativeLineNumber(), funcBody->GetSourceContextInfo(),

funcBody, false);

Assert(FAILED(hrParser) || funcBody->deferredParseNextFunctionId == nextFunctionId);

if (FAILED(hrParser))

{

hrParseCodeGen = MapDeferredReparseError(hrParser, \*se); // Map certain errors like OOM/SOE

AssertMsg(FAILED(hrParseCodeGen) && SUCCEEDED(hrParser), "Syntax errors should never be detected on deferred re-parse");

}

if (!SUCCEEDED(hrParser))

{

Throw::InternalError();

}

else if (!SUCCEEDED(hrParseCodeGen))

{

if (hrParseCodeGen == VBSERR\_OutOfStack)

{

Throw::StackOverflow(m\_scriptContext, nullptr);

}

else

{

Assert(hrParseCodeGen == E\_OUTOFMEMORY);

Throw::OutOfMemory();

}

}

UpdateFunctionBodyImpl(funcBody);

m\_hasBeenParsed = true;

returnFunctionBody = GetFunctionBody();

LEAVE\_PINNED\_SCOPE();

return returnFunctionBody;

}

#endif

void ParseableFunctionInfo::Finalize(bool isShutdown)

{

\_\_super::Finalize(isShutdown);

if (!this->m\_hasBeenParsed)

{

if (!this->GetSourceContextInfo()->IsDynamic() && !this->GetIsTopLevel())

{

this->m\_utf8SourceInfo->StopTrackingDeferredFunction(this->GetLocalFunctionId());

}

PERF\_COUNTER\_DEC(Code, DeferedFunction);

}

}

bool ParseableFunctionInfo::IsFakeGlobalFunc(ulong flags) const

{

return GetIsGlobalFunc() && !(flags & fscrGlobalCode);

}

bool ParseableFunctionInfo::GetExternalDisplaySourceName(BSTR\* sourceName)

{

Assert(sourceName);

if (IsDynamicScript() && GetUtf8SourceInfo()->GetDebugDocumentName(sourceName))

{

return true;

}

\*sourceName = ::SysAllocString(GetSourceName());

return \*sourceName != nullptr;

}

const wchar\_t\* FunctionProxy::WrapWithBrackets(const wchar\_t\* name, charcount\_t sz, ScriptContext\* scriptContext)

{

wchar\_t \* wrappedName = RecyclerNewArrayLeaf(scriptContext->GetRecycler(), wchar\_t, sz + 3); //[]\0

wrappedName[0] = L'[';

wchar\_t \*next = wrappedName;

js\_wmemcpy\_s(++next, sz, name, sz);

wrappedName[sz + 1] = L']';

wrappedName[sz + 2] = L'\0';

return wrappedName;

}

const wchar\_t\* FunctionProxy::GetShortDisplayName(charcount\_t \* shortNameLength)

{

const wchar\_t\* name = this->GetDisplayName();

uint nameLength = this->GetDisplayNameLength();

if (name == nullptr)

{

\*shortNameLength = 0;

return Constants::Empty;

}

if (IsConstantFunctionName(name))

{

\*shortNameLength = nameLength;

return name;

}

uint shortNameOffset = this->GetShortDisplayNameOffset();

const wchar\_t \* shortName = name + shortNameOffset;

bool isBracketCase = shortNameOffset != 0 && name[shortNameOffset-1] == '[';

Assert(nameLength >= shortNameOffset);

\*shortNameLength = nameLength - shortNameOffset;

if (!isBracketCase)

{

return shortName;

}

Assert(name[nameLength - 1] == ']');

wchar\_t \* finalshorterName = RecyclerNewArrayLeaf(this->GetScriptContext()->GetRecycler(), wchar\_t, \*shortNameLength);

js\_wmemcpy\_s(finalshorterName, \*shortNameLength, shortName, \*shortNameLength - 1); // we don't want the last character in shorterName

finalshorterName[\*shortNameLength - 1] = L'\0';

\*shortNameLength = \*shortNameLength - 1;

return finalshorterName;

}

/\*static\*/

bool FunctionProxy::IsConstantFunctionName(const wchar\_t\* srcName)

{

if (srcName == Js::Constants::GlobalFunction ||

srcName == Js::Constants::AnonymousFunction ||

srcName == Js::Constants::GlobalCode ||

srcName == Js::Constants::Anonymous ||

srcName == Js::Constants::UnknownScriptCode ||

srcName == Js::Constants::FunctionCode)

{

return true;

}

return false;

}

/\*static \*/

/\*Return value: Whether the target value is a recycler pointer or not\*/

bool FunctionProxy::SetDisplayName(const wchar\_t\* srcName, const wchar\_t\*\* destName, uint displayNameLength, ScriptContext \* scriptContext, SetDisplayNameFlags flags /\* default to None \*/)

{

Assert(destName);

Assert(scriptContext);

if (srcName == nullptr)

{

\*destName = (L"");

return false;

}

else if (IsConstantFunctionName(srcName) || (flags & SetDisplayNameFlagsDontCopy) != 0)

{

\*destName = srcName;

return (flags & SetDisplayNameFlagsRecyclerAllocated) != 0; // Return true if array is recycler allocated

}

else

{

uint numCharacters = displayNameLength + 1;

Assert((flags & SetDisplayNameFlagsDontCopy) == 0);

\*destName = RecyclerNewArrayLeaf(scriptContext->GetRecycler(), wchar\_t, numCharacters);

js\_wmemcpy\_s((wchar\_t \*)\*destName, numCharacters, srcName, numCharacters);

((wchar\_t \*)(\*destName))[numCharacters - 1] = L'\0';

return true;

}

}

void FunctionProxy::SetDisplayName(const wchar\_t\* srcName, WriteBarrierPtr<const wchar\_t>\* destName, uint displayNameLength, ScriptContext \* scriptContext, SetDisplayNameFlags flags /\* default to None \*/)

{

const wchar\_t\* dest = nullptr;

bool targetIsRecyclerMemory = SetDisplayName(srcName, &dest, displayNameLength, scriptContext, flags);

if (targetIsRecyclerMemory)

{

\*destName = dest;

}

else

{

destName->NoWriteBarrierSet(dest);

}

}

void ParseableFunctionInfo::SetDisplayName(const wchar\_t\* pszDisplayName)

{

size\_t len = wcslen(pszDisplayName);

if (len > UINT\_MAX)

{

// Can't support display name that big

Js::Throw::OutOfMemory();

}

SetDisplayName(pszDisplayName, (uint)len, 0);

}

void ParseableFunctionInfo::SetDisplayName(const wchar\_t\* pszDisplayName, uint displayNameLength, uint displayShortNameOffset, SetDisplayNameFlags flags /\* default to None \*/)

{

this->m\_displayNameLength = displayNameLength;

this->m\_displayShortNameOffset = displayShortNameOffset;

FunctionProxy::SetDisplayName(pszDisplayName, &this->m\_displayName, displayNameLength, m\_scriptContext, flags);

}

// SourceInfo methods

FunctionBody::StatementMapList \* FunctionBody::GetStatementMaps() const

{

return this->pStatementMaps;

}

/\* static \*/ FunctionBody::StatementMap \* FunctionBody::GetNextNonSubexpressionStatementMap(StatementMapList \*statementMapList, int & startingAtIndex)

{

AssertMsg(statementMapList != nullptr, "Must have valid statementMapList to execute");

FunctionBody::StatementMap \*map = statementMapList->Item(startingAtIndex);

while (map->isSubexpression && startingAtIndex < statementMapList->Count() - 1)

{

map = statementMapList->Item(++startingAtIndex);

}

if (map->isSubexpression) // Didn't find any non inner maps

{

return nullptr;

}

return map;

}

/\* static \*/ FunctionBody::StatementMap \* FunctionBody::GetPrevNonSubexpressionStatementMap(StatementMapList \*statementMapList, int & startingAtIndex)

{

AssertMsg(statementMapList != nullptr, "Must have valid statementMapList to execute");

FunctionBody::StatementMap \*map = statementMapList->Item(startingAtIndex);

while (startingAtIndex && map->isSubexpression)

{

map = statementMapList->Item(--startingAtIndex);

}

if (map->isSubexpression) // Didn't find any non inner maps

{

return nullptr;

}

return map;

}

void ParseableFunctionInfo::CloneSourceInfo(ScriptContext\* scriptContext, const ParseableFunctionInfo& other, ScriptContext\* othersScriptContext, uint sourceIndex)

{

if (!m\_utf8SourceHasBeenSet)

{

this->m\_utf8SourceInfo = scriptContext->GetSource(sourceIndex);

this->m\_sourceIndex = sourceIndex;

this->m\_cchStartOffset = other.m\_cchStartOffset;

this->m\_cchLength = other.m\_cchLength;

this->m\_lineNumber = other.m\_lineNumber;

this->m\_columnNumber = other.m\_columnNumber;

this->m\_isEval = other.m\_isEval;

this->m\_isDynamicFunction = other.m\_isDynamicFunction;

this->m\_cbStartOffset = other.StartOffset();

this->m\_cbLength = other.LengthInBytes();

this->m\_utf8SourceHasBeenSet = true;

if (this->IsFunctionBody())

{

this->GetFunctionBody()->FinishSourceInfo();

}

}

#if DBG

else

{

AssertMsg(this->m\_cchStartOffset == other.m\_cchStartOffset, "Mismatched source character start offset");

AssertMsg(this->StartOffset() == other.StartOffset(), "Mismatched source start offset");

AssertMsg(this->m\_cchLength == other.m\_cchLength, "Mismatched source character length");

AssertMsg(this->LengthInBytes() == other.LengthInBytes(), "Mismatch source byte length");

AssertMsg(this->GetUtf8SourceInfo()->GetSourceHolder() == scriptContext->GetSource(this->m\_sourceIndex)->GetSourceHolder(),

"Mismatched source holder pointer");

AssertMsg(this->m\_isEval == other.m\_isEval, "Mismatched source type");

AssertMsg(this->m\_isDynamicFunction == other.m\_isDynamicFunction, "Mismatch source type");

}

#endif

}

void ParseableFunctionInfo::SetSourceInfo(uint sourceIndex, ParseNodePtr node, bool isEval, bool isDynamicFunction)

{

if (!m\_utf8SourceHasBeenSet)

{

this->m\_sourceIndex = sourceIndex;

this->m\_cchStartOffset = node->ichMin;

this->m\_cchLength = node->LengthInCodepoints();

this->m\_lineNumber = node->sxFnc.lineNumber;

this->m\_columnNumber = node->sxFnc.columnNumber;

this->m\_isEval = isEval;

this->m\_isDynamicFunction = isDynamicFunction;

// It would have been better if we detect and reject large source buffer eariler before parsing

size\_t cbMin = node->sxFnc.cbMin;

size\_t lengthInBytes = node->sxFnc.LengthInBytes();

if (cbMin > UINT\_MAX || lengthInBytes > UINT\_MAX)

{

Js::Throw::OutOfMemory();

}

this->m\_cbStartOffset = (uint)cbMin;

this->m\_cbLength = (uint)lengthInBytes;

Assert(this->m\_utf8SourceInfo != nullptr);

this->m\_utf8SourceHasBeenSet = true;

if (this->IsFunctionBody())

{

this->GetFunctionBody()->FinishSourceInfo();

}

}

#if DBG

else

{

AssertMsg(this->m\_sourceIndex == sourceIndex, "Mismatched source index");

if (!this->GetIsGlobalFunc())

{

// In the global function case with a @cc\_on, we modify some of these values so it might

// not match on reparse (see ParseableFunctionInfo::Parse()).

AssertMsg(this->StartOffset() == node->sxFnc.cbMin, "Mismatched source start offset");

AssertMsg(this->m\_cchStartOffset == node->ichMin, "Mismatched source character start offset");

AssertMsg(this->m\_cchLength == node->LengthInCodepoints(), "Mismatched source length");

AssertMsg(this->LengthInBytes() == node->sxFnc.LengthInBytes(), "Mismatched source encoded byte length");

}

AssertMsg(this->m\_isEval == isEval, "Mismatched source type");

AssertMsg(this->m\_isDynamicFunction == isDynamicFunction, "Mismatch source type");

}

#endif

#if DBG\_DUMP

if (PHASE\_TRACE1(Js::FunctionSourceInfoParsePhase))

{

if (this->HasBody())

{

FunctionProxy\* proxy = this->GetFunctionProxy();

if (proxy->IsFunctionBody())

{

FunctionBody\* functionBody = this->GetFunctionBody();

Assert( functionBody != nullptr );

functionBody->PrintStatementSourceLineFromStartOffset(functionBody->StartInDocument());

Output::Flush();

}

}

}

#endif

}

bool FunctionBody::Is(void\* ptr)

{

if(!ptr)

{

return false;

}

return VirtualTableInfo<FunctionBody>::HasVirtualTable(ptr);

}

bool FunctionBody::HasLineBreak() const

{

return this->HasLineBreak(this->StartOffset(), this->m\_cchStartOffset + this->m\_cchLength);

}

bool FunctionBody::HasLineBreak(charcount\_t start, charcount\_t end) const

{

if (start > end) return false;

charcount\_t cchLength = end - start;

if (start < this->m\_cchStartOffset || cchLength > this->m\_cchLength) return false;

LPCUTF8 src = this->GetSource(L"FunctionBody::HasLineBreak");

LPCUTF8 last = src + this->LengthInBytes();

size\_t offset = this->LengthInBytes() == this->m\_cchLength ?

start - this->m\_cchStartOffset :

utf8::CharacterIndexToByteIndex(src, this->LengthInBytes(), start - this->m\_cchStartOffset, utf8::doAllowThreeByteSurrogates);

src = src + offset;

utf8::DecodeOptions options = utf8::doAllowThreeByteSurrogates;

for (charcount\_t cch = cchLength; cch > 0; --cch)

{

switch (utf8::Decode(src, last, options))

{

case '\r':

case '\n':

case 0x2028:

case 0x2029:

return true;

}

}

return false;

}

FunctionBody::StatementMap\* FunctionBody::GetMatchingStatementMapFromByteCode(int byteCodeOffset, bool ignoreSubexpressions /\* = false \*/)

{

StatementMapList \* pStatementMaps = this->GetStatementMaps();

if (pStatementMaps)

{

Assert(m\_sourceInfo.pSpanSequence == nullptr);

for (int index = 0; index < pStatementMaps->Count(); index++)

{

FunctionBody::StatementMap\* pStatementMap = pStatementMaps->Item(index);

if (!(ignoreSubexpressions && pStatementMap->isSubexpression) && pStatementMap->byteCodeSpan.Includes(byteCodeOffset))

{

return pStatementMap;

}

}

}

return nullptr;

}

// Returns the StatementMap for the offset.

// 1. Current statementMap if bytecodeoffset falls within bytecode's span

// 2. Previous if the bytecodeoffset is in between previous's end to current's begin

FunctionBody::StatementMap\* FunctionBody::GetEnclosingStatementMapFromByteCode(int byteCodeOffset, bool ignoreSubexpressions /\* = false \*/)

{

int index = GetEnclosingStatementIndexFromByteCode(byteCodeOffset, ignoreSubexpressions);

if (index != -1)

{

return this->GetStatementMaps()->Item(index);

}

return nullptr;

}

// Returns the index of StatementMap for

// 1. Current statementMap if bytecodeoffset falls within bytecode's span

// 2. Previous if the bytecodeoffset is in between previous's end to current's begin

// 3. -1 of the failures.

int FunctionBody::GetEnclosingStatementIndexFromByteCode(int byteCodeOffset, bool ignoreSubexpressions /\* = false \*/)

{

StatementMapList \* pStatementMaps = this->GetStatementMaps();

if (pStatementMaps == nullptr)

{

// e.g. internal library.

return -1;

}

Assert(m\_sourceInfo.pSpanSequence == nullptr);

for (int index = 0; index < pStatementMaps->Count(); index++)

{

FunctionBody::StatementMap\* pStatementMap = pStatementMaps->Item(index);

if (!(ignoreSubexpressions && pStatementMap->isSubexpression) && pStatementMap->byteCodeSpan.Includes(byteCodeOffset))

{

return index;

}

else if (!pStatementMap->isSubexpression && byteCodeOffset < pStatementMap->byteCodeSpan.begin) // We always ignore sub expressions when checking if we went too far

{

return index > 0 ? index - 1 : 0;

}

}

return pStatementMaps->Count() - 1;

}

// In some cases in legacy mode, due to the state scriptContext->windowIdList, the parser might not detect an eval call in the first parse but do so in the reparse

// This fixes up the state at the start of reparse

void FunctionBody::SaveState(ParseNodePtr pnode)

{

Assert(!this->IsReparsed());

this->SetChildCallsEval(!!pnode->sxFnc.ChildCallsEval());

this->SetCallsEval(!!pnode->sxFnc.CallsEval());

this->SetHasReferenceableBuiltInArguments(!!pnode->sxFnc.HasReferenceableBuiltInArguments());

}

void FunctionBody::RestoreState(ParseNodePtr pnode)

{

Assert(this->IsReparsed());

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

#endif

if(!!pnode->sxFnc.ChildCallsEval() != this->GetChildCallsEval())

{

OUTPUT\_VERBOSE\_TRACE(Js::DebuggerPhase, L"Child calls eval is different on debug reparse: %s(%s)\n", this->GetExternalDisplayName(), this->GetDebugNumberSet(debugStringBuffer));

}

if(!!pnode->sxFnc.CallsEval() != this->GetCallsEval())

{

OUTPUT\_VERBOSE\_TRACE(Js::DebuggerPhase, L"Calls eval is different on debug reparse: %s(%s)\n", this->GetExternalDisplayName(), this->GetDebugNumberSet(debugStringBuffer));

}

if(!!pnode->sxFnc.HasReferenceableBuiltInArguments() != this->HasReferenceableBuiltInArguments())

{

OUTPUT\_VERBOSE\_TRACE(Js::DebuggerPhase, L"Referencable Built in args is different on debug reparse: %s(%s)\n", this->GetExternalDisplayName(), this->GetDebugNumberSet(debugStringBuffer));

}

pnode->sxFnc.SetChildCallsEval(this->GetChildCallsEval());

pnode->sxFnc.SetCallsEval(this->GetCallsEval());

pnode->sxFnc.SetHasReferenceableBuiltInArguments(this->HasReferenceableBuiltInArguments());

}

// Retrieves statement map for given byte code offset.

// Parameters:

// - sourceOffset: byte code offset to get map for.

// - mapIndex: if not NULL, receives the index of found map.

FunctionBody::StatementMap\* FunctionBody::GetMatchingStatementMapFromSource(int sourceOffset, int\* pMapIndex /\* = nullptr \*/)

{

StatementMapList \* pStatementMaps = this->GetStatementMaps();

if (pStatementMaps && pStatementMaps->Count() > 0)

{

Assert(m\_sourceInfo.pSpanSequence == nullptr);

for (int index = pStatementMaps->Count() - 1; index >= 0; index--)

{

FunctionBody::StatementMap\* pStatementMap = pStatementMaps->Item(index);

if (!pStatementMap->isSubexpression && pStatementMap->sourceSpan.Includes(sourceOffset))

{

if (pMapIndex)

{

\*pMapIndex = index;

}

return pStatementMap;

}

}

}

if (pMapIndex)

{

\*pMapIndex = 0;

}

return nullptr;

}

//

// The function determine the line and column for a bytecode offset within the current script buffer.

//

bool FunctionBody::GetLineCharOffset(int byteCodeOffset, ULONG\* \_line, LONG\* \_charOffset, bool canAllocateLineCache /\*= true\*/)

{

Assert(!this->GetUtf8SourceInfo()->GetIsLibraryCode());

int startCharOfStatement = this->m\_cchStartOffset; // Default to the start of this function

if (m\_sourceInfo.pSpanSequence)

{

SmallSpanSequenceIter iter;

m\_sourceInfo.pSpanSequence->Reset(iter);

StatementData data;

if (m\_sourceInfo.pSpanSequence->GetMatchingStatementFromBytecode(byteCodeOffset, iter, data)

&& EndsAfter(data.sourceBegin))

{

startCharOfStatement = data.sourceBegin;

}

}

else

{

Js::FunctionBody::StatementMap\* map = this->GetEnclosingStatementMapFromByteCode(byteCodeOffset, false);

if (map && EndsAfter(map->sourceSpan.begin))

{

startCharOfStatement = map->sourceSpan.begin;

}

}

return this->GetLineCharOffsetFromStartChar(startCharOfStatement, \_line, \_charOffset, canAllocateLineCache);

}

bool FunctionBody::GetLineCharOffsetFromStartChar(int startCharOfStatement, ULONG\* \_line, LONG\* \_charOffset, bool canAllocateLineCache /\*= true\*/)

{

Assert(!this->GetUtf8SourceInfo()->GetIsLibraryCode());

// The following adjusts for where the script is within the document

ULONG line = this->GetHostStartLine();

charcount\_t column = 0;

ULONG lineCharOffset = 0;

charcount\_t lineByteOffset = 0;

if (startCharOfStatement > 0)

{

bool doSlowLookup = !canAllocateLineCache;

if (canAllocateLineCache)

{

HRESULT hr = m\_utf8SourceInfo->EnsureLineOffsetCacheNoThrow();

if (FAILED(hr))

{

if (hr != E\_OUTOFMEMORY)

{

Assert(hr == E\_ABORT); // The only other possible error we know about is ScriptAbort from QueryContinue.

return false;

}

// Clear the cache so it is not used.

this->m\_utf8SourceInfo->DeleteLineOffsetCache();

// We can try and do the slow lookup below

doSlowLookup = true;

}

}

charcount\_t cacheLine = 0;

this->m\_utf8SourceInfo->GetLineInfoForCharPosition(startCharOfStatement, &cacheLine, &column, &lineByteOffset, doSlowLookup);

// Update the tracking variables to jump to the line position (only need to jump if not on the first line).

if (cacheLine > 0)

{

line += cacheLine;

lineCharOffset = startCharOfStatement - column;

}

}

if (this->GetSourceContextInfo()->IsDynamic() && this->m\_isDynamicFunction)

{

line -= JavascriptFunction::numberLinesPrependedToAnonymousFunction;

}

if(\_line)

{

\*\_line = line;

}

if(\_charOffset)

{

\*\_charOffset = column;

// If we are at the beginning of the host code, adjust the offset based on the host provided offset

if (this->GetHostSrcInfo()->dlnHost == line)

{

\*\_charOffset += (LONG)this->GetHostStartColumn();

}

}

return true;

}

bool FunctionBody::GetStatementIndexAndLengthAt(int byteCodeOffset, UINT32\* statementIndex, UINT32\* statementLength)

{

Assert(statementIndex != nullptr);

Assert(statementLength != nullptr);

Assert(m\_scriptContext->IsInDebugMode());

StatementMap \* statement = GetEnclosingStatementMapFromByteCode(byteCodeOffset, false);

Assert(statement != nullptr);

// Bailout if we are unable to find a statement.

// We shouldn't be missing these when a debugger is attached but we don't want to AV on retail builds.

if (statement == nullptr)

{

return false;

}

Assert(m\_utf8SourceInfo);

const SRCINFO \* srcInfo = m\_utf8SourceInfo->GetSrcInfo();

// Offset from the beginning of the document minus any host-supplied source characters.

// Host supplied characters are inserted (for example) around onload:

// onload="foo('somestring', 0)" -> function onload(event).{.foo('somestring', 0).}

ULONG offsetFromDocumentBegin = srcInfo ? srcInfo->ulCharOffset - srcInfo->ichMinHost : 0;

\*statementIndex = statement->sourceSpan.Begin() + offsetFromDocumentBegin;

\*statementLength = statement->sourceSpan.End() - statement->sourceSpan.Begin();

return true;

}

void FunctionBody::RecordFrameDisplayRegister(RegSlot slot)

{

AssertMsg(slot != 0, "The assumption that the Frame Display Register cannot be at the 0 slot is wrong.");

SetFrameDisplayRegister(slot);

}

void FunctionBody::RecordObjectRegister(RegSlot slot)

{

AssertMsg(slot != 0, "The assumption that the Object Register cannot be at the 0 slot is wrong.");

SetObjectRegister(slot);

}

Js::RootObjectBase \* FunctionBody::GetRootObject() const

{

// Safe to be used by the JIT thread

Assert(this->m\_constTable != nullptr);

return (Js::RootObjectBase \*)this->m\_constTable[Js::FunctionBody::RootObjectRegSlot - FunctionBody::FirstRegSlot];

}

Js::RootObjectBase \* FunctionBody::LoadRootObject() const

{

if (this->GetModuleID() == kmodGlobal)

{

return JavascriptOperators::OP\_LdRoot(this->GetScriptContext());

}

return JavascriptOperators::GetModuleRoot(this->GetModuleID(), this->GetScriptContext());

}

#if ENABLE\_NATIVE\_CODEGEN

FunctionEntryPointInfo \* FunctionBody::GetEntryPointFromNativeAddress(DWORD\_PTR codeAddress)

{

FunctionEntryPointInfo \* entryPoint = nullptr;

this->MapEntryPoints([&entryPoint, &codeAddress](int index, FunctionEntryPointInfo \* currentEntryPoint)

{

// We need to do a second check for IsNativeCode because the entry point could be in the process of

// being recorded on the background thread

if (currentEntryPoint->IsInNativeAddressRange(codeAddress))

{

entryPoint = currentEntryPoint;

}

});

return entryPoint;

}

LoopEntryPointInfo \* FunctionBody::GetLoopEntryPointInfoFromNativeAddress(DWORD\_PTR codeAddress, uint loopNum) const

{

LoopEntryPointInfo \* entryPoint = nullptr;

LoopHeader \* loopHeader = this->GetLoopHeader(loopNum);

Assert(loopHeader);

loopHeader->MapEntryPoints([&](int index, LoopEntryPointInfo \* currentEntryPoint)

{

if (currentEntryPoint->IsCodeGenDone() &&

codeAddress >= currentEntryPoint->GetNativeAddress() &&

codeAddress < currentEntryPoint->GetNativeAddress() + currentEntryPoint->GetCodeSize())

{

entryPoint = currentEntryPoint;

}

});

return entryPoint;

}

int FunctionBody::GetStatementIndexFromNativeOffset(SmallSpanSequence \*pThrowSpanSequence, uint32 nativeOffset)

{

int statementIndex = -1;

if (pThrowSpanSequence)

{

SmallSpanSequenceIter iter;

StatementData tmpData;

if (pThrowSpanSequence->GetMatchingStatementFromBytecode(nativeOffset, iter, tmpData))

{

statementIndex = tmpData.sourceBegin; // sourceBegin represents statementIndex here

}

else

{

// If nativeOffset falls on the last span, GetMatchingStatement would miss it because SmallSpanSequence

// does not know about the last span end. Since we checked that codeAddress is within our range,

// we can safely consider it matches the last span.

statementIndex = iter.accumulatedSourceBegin;

}

}

return statementIndex;

}

int FunctionBody::GetStatementIndexFromNativeAddress(SmallSpanSequence \*pThrowSpanSequence, DWORD\_PTR codeAddress, DWORD\_PTR nativeBaseAddress)

{

uint32 nativeOffset = (uint32)(codeAddress - nativeBaseAddress);

return GetStatementIndexFromNativeOffset(pThrowSpanSequence, nativeOffset);

}

#endif

BOOL FunctionBody::GetMatchingStatementMap(StatementData &data, int statementIndex, FunctionBody \*inlinee)

{

SourceInfo \*si = &this->m\_sourceInfo;

if (inlinee)

{

si = &inlinee->m\_sourceInfo;

Assert(si);

}

if (statementIndex >= 0)

{

SmallSpanSequence \*pSpanSequence = si->pSpanSequence;

if (pSpanSequence)

{

SmallSpanSequenceIter iter;

pSpanSequence->Reset(iter);

if (pSpanSequence->Item(statementIndex, iter, data))

{

return TRUE;

}

}

else

{

StatementMapList\* pStatementMaps = GetStatementMaps();

Assert(pStatementMaps);

if (statementIndex >= pStatementMaps->Count())

{

return FALSE;

}

data.sourceBegin = pStatementMaps->Item(statementIndex)->sourceSpan.begin;

data.bytecodeBegin = pStatementMaps->Item(statementIndex)->byteCodeSpan.begin;

return TRUE;

}

}

return FALSE;

}

void FunctionBody::FindClosestStatements(long characterOffset, StatementLocation \*firstStatementLocation, StatementLocation \*secondStatementLocation)

{

auto statementMaps = this->GetStatementMaps();

if (statementMaps)

{

for(int i = 0; i < statementMaps->Count(); i++)

{

regex::Interval\* pSourceSpan = &(statementMaps->Item(i)->sourceSpan);

if (FunctionBody::IsDummyGlobalRetStatement(pSourceSpan))

{

// Workaround for handling global return, which is a empty range.

continue;

}

if (pSourceSpan->begin < characterOffset

&& (firstStatementLocation->function == nullptr || firstStatementLocation->statement.begin < pSourceSpan->begin))

{

firstStatementLocation->function = this;

firstStatementLocation->statement = \*pSourceSpan;

firstStatementLocation->bytecodeSpan = statementMaps->Item(i)->byteCodeSpan;

}

else if (pSourceSpan->begin >= characterOffset

&& (secondStatementLocation->function == nullptr || secondStatementLocation->statement.begin > pSourceSpan->begin))

{

secondStatementLocation->function = this;

secondStatementLocation->statement = \*pSourceSpan;

secondStatementLocation->bytecodeSpan = statementMaps->Item(i)->byteCodeSpan;

}

}

}

}

#if ENABLE\_NATIVE\_CODEGEN

BOOL FunctionBody::GetMatchingStatementMapFromNativeAddress(DWORD\_PTR codeAddress, StatementData &data, uint loopNum, FunctionBody \*inlinee /\* = nullptr \*/)

{

SmallSpanSequence \* spanSequence = nullptr;

DWORD\_PTR nativeBaseAddress = NULL;

EntryPointInfo \* entryPoint;

if (loopNum == -1)

{

entryPoint = GetEntryPointFromNativeAddress(codeAddress);

}

else

{

entryPoint = GetLoopEntryPointInfoFromNativeAddress(codeAddress, loopNum);

}

if (entryPoint != nullptr)

{

spanSequence = entryPoint->GetNativeThrowSpanSequence();

nativeBaseAddress = entryPoint->GetNativeAddress();

}

int statementIndex = GetStatementIndexFromNativeAddress(spanSequence, codeAddress, nativeBaseAddress);

return GetMatchingStatementMap(data, statementIndex, inlinee);

}

BOOL FunctionBody::GetMatchingStatementMapFromNativeOffset(DWORD\_PTR codeAddress, uint32 offset, StatementData &data, uint loopNum, FunctionBody \*inlinee /\* = nullptr \*/)

{

EntryPointInfo \* entryPoint;

if (loopNum == -1)

{

entryPoint = GetEntryPointFromNativeAddress(codeAddress);

}

else

{

entryPoint = GetLoopEntryPointInfoFromNativeAddress(codeAddress, loopNum);

}

SmallSpanSequence \*spanSequence = entryPoint ? entryPoint->GetNativeThrowSpanSequence() : nullptr;

int statementIndex = GetStatementIndexFromNativeOffset(spanSequence, offset);

return GetMatchingStatementMap(data, statementIndex, inlinee);

}

#endif

#if ENABLE\_PROFILE\_INFO

void FunctionBody::LoadDynamicProfileInfo()

{

SourceDynamicProfileManager \* sourceDynamicProfileManager = GetSourceContextInfo()->sourceDynamicProfileManager;

if (sourceDynamicProfileManager != nullptr)

{

this->dynamicProfileInfo = sourceDynamicProfileManager->GetDynamicProfileInfo(this);

#if DBG\_DUMP

if(this->dynamicProfileInfo)

{

if (Configuration::Global.flags.Dump.IsEnabled(DynamicProfilePhase, this->GetSourceContextId(), this->GetLocalFunctionId()))

{

Output::Print(L"Loaded:");

this->dynamicProfileInfo->Dump(this);

}

}

#endif

}

#ifdef DYNAMIC\_PROFILE\_MUTATOR

DynamicProfileMutator::Mutate(this);

#endif

}

bool FunctionBody::NeedEnsureDynamicProfileInfo() const

{

// Only need to ensure dynamic profile if we don't already have link up the dynamic profile info

// and dynamic profile collection is enabled

return

!this->m\_isFromNativeCodeModule &&

!this->m\_isAsmJsFunction &&

!this->HasExecutionDynamicProfileInfo() &&

DynamicProfileInfo::IsEnabled(this);

}

DynamicProfileInfo \* FunctionBody::EnsureDynamicProfileInfo()

{

if (this->NeedEnsureDynamicProfileInfo())

{

m\_scriptContext->AddDynamicProfileInfo(this, &this->dynamicProfileInfo);

Assert(!this->HasExecutionDynamicProfileInfo());

this->hasExecutionDynamicProfileInfo = true;

}

return this->dynamicProfileInfo;

}

DynamicProfileInfo\* FunctionBody::AllocateDynamicProfile()

{

return DynamicProfileInfo::New(m\_scriptContext->GetRecycler(), this);

}

#endif

BOOL FunctionBody::IsNativeOriginalEntryPoint() const

{

#if ENABLE\_NATIVE\_CODEGEN

return IsNativeFunctionAddr(this->GetScriptContext(), this->originalEntryPoint);

#else

return false;

#endif

}

bool FunctionBody::IsSimpleJitOriginalEntryPoint() const

{

const FunctionEntryPointInfo \*const simpleJitEntryPointInfo = GetSimpleJitEntryPointInfo();

return

simpleJitEntryPointInfo &&

reinterpret\_cast<Js::JavascriptMethod>(simpleJitEntryPointInfo->GetNativeAddress()) == originalEntryPoint;

}

void FunctionProxy::Finalize(bool isShutdown)

{

\_\_super::Finalize(isShutdown);

this->CleanupFunctionProxyCounters();

}

#if DBG

bool FunctionBody::HasValidSourceInfo()

{

SourceContextInfo\* sourceContextInfo;

if (m\_scriptContext->GetSourceContextInfoMap())

{

if(m\_scriptContext->GetSourceContextInfoMap()->TryGetValue(this->GetHostSourceContext(), &sourceContextInfo) &&

sourceContextInfo == this->GetSourceContextInfo())

{

return true;

}

}

Assert(this->IsDynamicScript());

if(m\_scriptContext->GetDynamicSourceContextInfoMap())

{

if(m\_scriptContext->GetDynamicSourceContextInfoMap()->TryGetValue(this->GetSourceContextInfo()->hash, &sourceContextInfo) &&

sourceContextInfo == this->GetSourceContextInfo())

{

return true;

}

}

// The SourceContextInfo will not be added to the dynamicSourceContextInfoMap, if they are host provided dynamic code. But they are valid source context info

if (this->GetSourceContextInfo()->isHostDynamicDocument)

{

return true;

}

return m\_scriptContext->IsNoContextSourceContextInfo(this->GetSourceContextInfo());

}

// originalEntryPoint: DefaultDeferredParsingThunk, DefaultDeferredDeserializeThunk, DefaultEntryThunk, dynamic interpreter thunk or native entry point

// directEntryPoint:

// if (!profiled) - DefaultDeferredParsingThunk, DefaultDeferredDeserializeThunk, DefaultEntryThunk, CheckCodeGenThunk,

// dynamic interpreter thunk, native entry point

// if (profiling) - ProfileDeferredParsingThunk, ProfileDeferredDeserializeThunk, ProfileEntryThunk, CheckCodeGenThunk

bool FunctionProxy::HasValidNonProfileEntryPoint() const

{

JavascriptMethod directEntryPoint = (JavascriptMethod)this->GetDefaultEntryPointInfo()->address;

JavascriptMethod originalEntryPoint = this->originalEntryPoint;

// Check the direct entry point to see if it is codegen thunk

// if it is not, the background codegen thread has updated both original entry point and direct entry point

// and they should still match, same as cases other then code gen

return IsIntermediateCodeGenThunk(directEntryPoint) || originalEntryPoint == directEntryPoint

#if ENABLE\_PROFILE\_INFO

|| (directEntryPoint == DynamicProfileInfo::EnsureDynamicProfileInfoThunk &&

this->IsFunctionBody() && this->GetFunctionBody()->IsNativeOriginalEntryPoint()

#ifdef ASMJS\_PLAT

|| (GetFunctionBody()->GetIsAsmJsFunction() && directEntryPoint == AsmJsDefaultEntryThunk)

|| (IsAsmJsCodeGenThunk(directEntryPoint))

#endif

);

#endif

;

}

bool FunctionProxy::HasValidProfileEntryPoint() const

{

JavascriptMethod directEntryPoint = (JavascriptMethod)this->GetDefaultEntryPointInfo()->address;

if (this->originalEntryPoint == DefaultDeferredParsingThunk)

{

return directEntryPoint == ProfileDeferredParsingThunk;

}

if (this->originalEntryPoint == DefaultDeferredDeserializeThunk)

{

return directEntryPoint == ProfileDeferredDeserializeThunk;

}

if (!this->IsFunctionBody())

{

return false;

}

#if ENABLE\_PROFILE\_INFO

FunctionBody \* functionBody = this->GetFunctionBody();

if (functionBody->IsInterpreterThunk() || functionBody->IsSimpleJitOriginalEntryPoint())

{

return directEntryPoint == ProfileEntryThunk || IsIntermediateCodeGenThunk(directEntryPoint);

}

#if ENABLE\_NATIVE\_CODEGEN

// In the profiler mode, the EnsureDynamicProfileInfoThunk is valid as we would be assigning to appropriate thunk when that thunk called.

return functionBody->IsNativeOriginalEntryPoint() &&

(directEntryPoint == DynamicProfileInfo::EnsureDynamicProfileInfoThunk || directEntryPoint == ProfileEntryThunk);

#endif

#else

return true;

#endif

}

bool FunctionProxy::HasValidEntryPoint() const

{

if (!m\_scriptContext->HadProfiled() &&

!(m\_scriptContext->IsInDebugMode() && m\_scriptContext->IsExceptionWrapperForBuiltInsEnabled()))

{

return this->HasValidNonProfileEntryPoint();

}

if (m\_scriptContext->IsProfiling())

{

return this->HasValidProfileEntryPoint();

}

return this->HasValidNonProfileEntryPoint() || this->HasValidProfileEntryPoint();

}

#endif

void ParseableFunctionInfo::SetDeferredParsingEntryPoint()

{

Assert(m\_scriptContext->DeferredParsingThunk == ProfileDeferredParsingThunk

|| m\_scriptContext->DeferredParsingThunk == DefaultDeferredParsingThunk);

this->SetEntryPoint(this->GetDefaultEntryPointInfo(), m\_scriptContext->DeferredParsingThunk);

originalEntryPoint = DefaultDeferredParsingThunk;

}

void ParseableFunctionInfo::SetInitialDefaultEntryPoint()

{

Assert(m\_scriptContext->CurrentThunk == ProfileEntryThunk || m\_scriptContext->CurrentThunk == DefaultEntryThunk);

Assert(originalEntryPoint == DefaultDeferredParsingThunk || originalEntryPoint == ProfileDeferredParsingThunk ||

originalEntryPoint == DefaultDeferredDeserializeThunk || originalEntryPoint == ProfileDeferredDeserializeThunk ||

originalEntryPoint == DefaultEntryThunk || originalEntryPoint == ProfileEntryThunk);

Assert(this->m\_defaultEntryPointInfo != nullptr);

// CONSIDER: we can optimize this to generate the dynamic interpreter thunk up front

// If we know that we are in the defer parsing thunk already

this->SetEntryPoint(this->GetDefaultEntryPointInfo(), m\_scriptContext->CurrentThunk);

this->originalEntryPoint = DefaultEntryThunk;

}

void FunctionBody::SetCheckCodeGenEntryPoint(FunctionEntryPointInfo\* entryPointInfo, JavascriptMethod entryPoint)

{

Assert(IsIntermediateCodeGenThunk(entryPoint));

Assert(

this->GetEntryPoint(entryPointInfo) == m\_scriptContext->CurrentThunk ||

(entryPointInfo == this->m\_defaultEntryPointInfo && this->IsInterpreterThunk()) ||

(

GetSimpleJitEntryPointInfo() &&

GetEntryPoint(entryPointInfo) == reinterpret\_cast<void \*>(GetSimpleJitEntryPointInfo()->GetNativeAddress())

));

this->SetEntryPoint(entryPointInfo, entryPoint);

}

#if DYNAMIC\_INTERPRETER\_THUNK

void FunctionBody::GenerateDynamicInterpreterThunk()

{

if (this->m\_dynamicInterpreterThunk == nullptr)

{

// NOTE: Etw rundown thread may be reading this->dynamicInterpreterThunk concurrently. We don't need to synchronize

// access as it is ok for etw rundown to get either null or updated new value.

if (m\_isAsmJsFunction)

{

this->originalEntryPoint = this->m\_scriptContext->GetNextDynamicAsmJsInterpreterThunk(&this->m\_dynamicInterpreterThunk);

}

else

{

this->originalEntryPoint = this->m\_scriptContext->GetNextDynamicInterpreterThunk(&this->m\_dynamicInterpreterThunk);

}

JS\_ETW(EtwTrace::LogMethodInterpreterThunkLoadEvent(this));

}

else

{

this->originalEntryPoint = (JavascriptMethod)InterpreterThunkEmitter::ConvertToEntryPoint(this->m\_dynamicInterpreterThunk);

}

}

JavascriptMethod FunctionBody::EnsureDynamicInterpreterThunk(FunctionEntryPointInfo\* entryPointInfo)

{

// This may be first call to the function, make sure we have dynamic profile info

//

// We need to ensure dynamic profile info even if we didn't generate a dynamic interpreter thunk

// This happens when we go through CheckCodeGen thunk, to DelayDynamicInterpreterThunk, to here

// but the background codegen thread updated the entry point with the native entry point.

this->EnsureDynamicProfileInfo();

Assert(HasValidEntryPoint());

if (InterpreterStackFrame::IsDelayDynamicInterpreterThunk(this->GetEntryPoint(entryPointInfo)))

{

// We are not doing code gen on this function, just change the entry point directly

Assert(InterpreterStackFrame::IsDelayDynamicInterpreterThunk(originalEntryPoint));

GenerateDynamicInterpreterThunk();

this->SetEntryPoint(entryPointInfo, originalEntryPoint);

}

else if (this->GetEntryPoint(entryPointInfo) == ProfileEntryThunk)

{

// We are not doing codegen on this function, just change the entry point directly

// Don't replace the profile entry thunk

Assert(InterpreterStackFrame::IsDelayDynamicInterpreterThunk(originalEntryPoint));

GenerateDynamicInterpreterThunk();

}

else if (InterpreterStackFrame::IsDelayDynamicInterpreterThunk(originalEntryPoint))

{

JsUtil::JobProcessor \* jobProcessor = this->GetScriptContext()->GetThreadContext()->GetJobProcessor();

if (jobProcessor->ProcessesInBackground())

{

JsUtil::BackgroundJobProcessor \* backgroundJobProcessor = static\_cast<JsUtil::BackgroundJobProcessor \*>(jobProcessor);

AutoCriticalSection autocs(backgroundJobProcessor->GetCriticalSection());

// Check again under lock

if (InterpreterStackFrame::IsDelayDynamicInterpreterThunk(originalEntryPoint))

{

// If the original entry point is DelayDynamicInterpreterThunk then there must be a version of this

// function being codegen'd.

Assert(IsIntermediateCodeGenThunk((JavascriptMethod)this->GetEntryPoint(this->GetDefaultEntryPointInfo())) || IsAsmJsCodeGenThunk((JavascriptMethod)this->GetEntryPoint(this->GetDefaultEntryPointInfo())));

GenerateDynamicInterpreterThunk();

}

}

else

{

// If the original entry point is DelayDynamicInterpreterThunk then there must be a version of this

// function being codegen'd.

Assert(IsIntermediateCodeGenThunk((JavascriptMethod)this->GetEntryPoint(this->GetDefaultEntryPointInfo())) || IsAsmJsCodeGenThunk((JavascriptMethod)this->GetEntryPoint(this->GetDefaultEntryPointInfo())));

GenerateDynamicInterpreterThunk();

}

}

return this->originalEntryPoint;

}

#endif

#if ENABLE\_NATIVE\_CODEGEN

void FunctionBody::SetNativeEntryPoint(FunctionEntryPointInfo\* entryPointInfo, JavascriptMethod originalEntryPoint, Var directEntryPoint)

{

if(entryPointInfo->nativeEntryPointProcessed)

{

return;

}

bool isAsmJs = this->GetIsAsmjsMode();

Assert(IsIntermediateCodeGenThunk((JavascriptMethod)entryPointInfo->address) || CONFIG\_FLAG(Prejit) || this->m\_isFromNativeCodeModule || isAsmJs);

entryPointInfo->EnsureIsReadyToCall();

// keep originalEntryPoint updated with the latest known good native entry point

if (entryPointInfo == this->GetDefaultEntryPointInfo())

{

this->originalEntryPoint = originalEntryPoint;

}

if (entryPointInfo->entryPointIndex == 0 && this->NeedEnsureDynamicProfileInfo())

{

entryPointInfo->address = DynamicProfileInfo::EnsureDynamicProfileInfoThunk;

}

else

{

entryPointInfo->address = directEntryPoint;

}

if (isAsmJs)

{

// release the old entrypointinfo if available

FunctionEntryPointInfo\* oldEntryPointInfo = entryPointInfo->GetOldFunctionEntryPointInfo();

if (oldEntryPointInfo)

{

this->GetScriptContext()->GetThreadContext()->QueueFreeOldEntryPointInfoIfInScript(oldEntryPointInfo);

oldEntryPointInfo = nullptr;

}

}

this->CaptureDynamicProfileState(entryPointInfo);

if(entryPointInfo->GetJitMode() == ExecutionMode::SimpleJit)

{

Assert(GetExecutionMode() == ExecutionMode::SimpleJit);

SetSimpleJitEntryPointInfo(entryPointInfo);

ResetSimpleJitCallCount();

}

else

{

Assert(entryPointInfo->GetJitMode() == ExecutionMode::FullJit);

Assert(isAsmJs || GetExecutionMode() == ExecutionMode::FullJit);

entryPointInfo->callsCount =

static\_cast<uint8>(

min(

static\_cast<uint>(static\_cast<uint8>(CONFIG\_FLAG(MinBailOutsBeforeRejit))) \*

(Js::FunctionEntryPointInfo::GetDecrCallCountPerBailout() - 1),

0xffu));

}

TraceExecutionMode();

if(entryPointInfo->GetJitMode() == ExecutionMode::SimpleJit)

{

Assert(GetExecutionMode() == ExecutionMode::SimpleJit);

SetSimpleJitEntryPointInfo(entryPointInfo);

ResetSimpleJitCallCount();

}

else

{

Assert(entryPointInfo->GetJitMode() == ExecutionMode::FullJit);

Assert(GetExecutionMode() == ExecutionMode::FullJit);

entryPointInfo->callsCount =

static\_cast<uint8>(

min(

static\_cast<uint>(static\_cast<uint8>(CONFIG\_FLAG(MinBailOutsBeforeRejit))) \*

(Js::FunctionEntryPointInfo::GetDecrCallCountPerBailout() - 1),

0xffu));

}

JS\_ETW(EtwTrace::LogMethodNativeLoadEvent(this, entryPointInfo));

#ifdef \_M\_ARM

// For ARM we need to make sure that pipeline is synchronized with memory/cache for newly jitted code.

\_InstructionSynchronizationBarrier();

#endif

entryPointInfo->nativeEntryPointProcessed = true;

}

void FunctionBody::DefaultSetNativeEntryPoint(FunctionEntryPointInfo\* entryPointInfo, FunctionBody \* functionBody, JavascriptMethod entryPoint)

{

Assert(functionBody->m\_scriptContext->CurrentThunk == DefaultEntryThunk);

functionBody->SetNativeEntryPoint(entryPointInfo, entryPoint, entryPoint);

}

void FunctionBody::ProfileSetNativeEntryPoint(FunctionEntryPointInfo\* entryPointInfo, FunctionBody \* functionBody, JavascriptMethod entryPoint)

{

Assert(functionBody->m\_scriptContext->CurrentThunk == ProfileEntryThunk);

functionBody->SetNativeEntryPoint(entryPointInfo, entryPoint, ProfileEntryThunk);

}

Js::JavascriptMethod FunctionBody::GetLoopBodyEntryPoint(Js::LoopHeader \* loopHeader, int entryPointIndex)

{

#if DBG

this->GetLoopNumber(loopHeader);

#endif

return (Js::JavascriptMethod)(loopHeader->GetEntryPointInfo(entryPointIndex)->address);

}

void FunctionBody::SetLoopBodyEntryPoint(Js::LoopHeader \* loopHeader, EntryPointInfo\* entryPointInfo, Js::JavascriptMethod entryPoint)

{

#if DBG\_DUMP

uint loopNum = this->GetLoopNumber(loopHeader);

if (PHASE\_TRACE1(Js::JITLoopBodyPhase))

{

DumpFunctionId(true);

Output::Print(L": %-20s LoopBody EntryPt Loop: %2d Address : %x\n", GetDisplayName(), loopNum, entryPoint);

Output::Flush();

}

#endif

Assert(((LoopEntryPointInfo\*) entryPointInfo)->loopHeader == loopHeader);

Assert(entryPointInfo->address == nullptr);

entryPointInfo->address = (void\*)entryPoint;

// reset the counter to 1 less than the threshold for TJLoopBody

if (loopHeader->GetCurrentEntryPointInfo()->GetIsAsmJSFunction())

{

loopHeader->interpretCount = entryPointInfo->GetFunctionBody()->GetLoopInterpretCount(loopHeader) - 1;

}

JS\_ETW(EtwTrace::LogLoopBodyLoadEvent(this, loopHeader, ((LoopEntryPointInfo\*) entryPointInfo)));

}

#endif

void FunctionBody::MarkScript(ByteBlock \*byteCodeBlock, ByteBlock\* auxBlock, ByteBlock\* auxContextBlock,

uint byteCodeCount, uint byteCodeInLoopCount, uint byteCodeWithoutLDACount)

{

CheckNotExecuting();

CheckEmpty();

#ifdef PERF\_COUNTERS

DWORD byteCodeSize = byteCodeBlock->GetLength()

+ (auxBlock? auxBlock->GetLength() : 0)

+ (auxContextBlock? auxContextBlock->GetLength() : 0);

PERF\_COUNTER\_ADD(Code, DynamicByteCodeSize, byteCodeSize);

#endif

m\_byteCodeCount = byteCodeCount;

m\_byteCodeInLoopCount = byteCodeInLoopCount;

m\_byteCodeWithoutLDACount = byteCodeWithoutLDACount;

InitializeExecutionModeAndLimits();

this->auxBlock = auxBlock;

this->auxContextBlock = auxContextBlock;

// Memory barrier needed here to make sure the background codegen thread's inliner

// gets all the assignment before it sees that the function has been parse

MemoryBarrier();

this->byteCodeBlock = byteCodeBlock;

PERF\_COUNTER\_ADD(Code, TotalByteCodeSize, byteCodeSize);

// If this is a defer parse function body, we would not have registered it

// on the function bodies list so we should register it now

if (!this->m\_isFuncRegistered)

{

this->m\_utf8SourceInfo->SetFunctionBody(this);

}

}

uint

FunctionBody::GetLoopNumber(LoopHeader const \* loopHeader) const

{

Assert(loopHeader >= this->loopHeaderArray);

uint loopNum = (uint)(loopHeader - this->loopHeaderArray);

Assert(loopNum < GetLoopCount());

return loopNum;

}

bool FunctionBody::InstallProbe(int offset)

{

if (offset < 0 || ((uint)offset + 1) >= byteCodeBlock->GetLength())

{

return false;

}

byte\* pbyteCodeBlockBuffer = this->byteCodeBlock->GetBuffer();

if(!GetProbeBackingBlock())

{

// The probe backing block is set on a different thread than the main thread

// The recycler doesn't like allocations from a different thread, so we allocate

// the backing byte code block in the arena

ArenaAllocator \*pArena = m\_scriptContext->AllocatorForDiagnostics();

AssertMem(pArena);

ByteBlock\* probeBackingBlock = ByteBlock::NewFromArena(pArena, pbyteCodeBlockBuffer, byteCodeBlock->GetLength());

SetProbeBackingBlock(probeBackingBlock);

}

// Make sure Break opcode only need one byte

Assert(OpCodeUtil::IsSmallEncodedOpcode(OpCode::Break));

#if ENABLE\_NATIVE\_CODEGEN

Assert(!OpCodeAttr::HasMultiSizeLayout(OpCode::Break));

#endif

\*(byte \*)(pbyteCodeBlockBuffer + offset) = (byte)OpCode::Break;

++m\_sourceInfo.m\_probeCount;

return true;

}

bool FunctionBody::UninstallProbe(int offset)

{

if (offset < 0 || ((uint)offset + 1) >= byteCodeBlock->GetLength())

{

return false;

}

byte\* pbyteCodeBlockBuffer = byteCodeBlock->GetBuffer();

Js::OpCode originalOpCode = ByteCodeReader::PeekByteOp(GetProbeBackingBlock()->GetBuffer() + offset);

\*(pbyteCodeBlockBuffer + offset) = (byte)originalOpCode;

--m\_sourceInfo.m\_probeCount;

AssertMsg(m\_sourceInfo.m\_probeCount >= 0, "Probe (Break Point) count became negative!");

return true;

}

bool FunctionBody::ProbeAtOffset(int offset, OpCode\* pOriginalOpcode)

{

if (!GetProbeBackingBlock())

{

return false;

}

if (offset < 0 || ((uint)offset + 1) >= this->byteCodeBlock->GetLength())

{

AssertMsg(false, "ProbeAtOffset called with out of bounds offset");

return false;

}

Js::OpCode runningOpCode = ByteCodeReader::PeekByteOp(this->byteCodeBlock->GetBuffer() + offset);

Js::OpCode originalOpcode = ByteCodeReader::PeekByteOp(GetProbeBackingBlock()->GetBuffer() + offset);

if ( runningOpCode != originalOpcode)

{

\*pOriginalOpcode = originalOpcode;

return true;

}

else

{

// e.g. inline break or a step hit and is checking for a bp

return false;

}

}

void FunctionBody::CloneByteCodeInto(ScriptContext \* scriptContext, FunctionBody \*newFunctionBody, uint sourceIndex)

{

((ParseableFunctionInfo\*) this)->CopyFunctionInfoInto(scriptContext, newFunctionBody, sourceIndex);

newFunctionBody->m\_constCount = this->m\_constCount;

newFunctionBody->m\_varCount = this->m\_varCount;

newFunctionBody->m\_outParamMaxDepth = this->m\_outParamMaxDepth;

newFunctionBody->m\_firstTmpReg = this->m\_firstTmpReg;

newFunctionBody->localClosureRegister = this->localClosureRegister;

newFunctionBody->localFrameDisplayRegister = this->localFrameDisplayRegister;

newFunctionBody->envRegister = this->envRegister;

newFunctionBody->thisRegisterForEventHandler = this->thisRegisterForEventHandler;

newFunctionBody->firstInnerScopeRegister = this->firstInnerScopeRegister;

newFunctionBody->funcExprScopeRegister = this->funcExprScopeRegister;

newFunctionBody->innerScopeCount = this->innerScopeCount;

newFunctionBody->hasCachedScopePropIds = this->hasCachedScopePropIds;

newFunctionBody->loopCount = this->loopCount;

newFunctionBody->profiledDivOrRemCount = this->profiledDivOrRemCount;

newFunctionBody->profiledSwitchCount = this->profiledSwitchCount;

newFunctionBody->profiledCallSiteCount = this->profiledCallSiteCount;

newFunctionBody->profiledArrayCallSiteCount = this->profiledArrayCallSiteCount;

newFunctionBody->profiledReturnTypeCount = this->profiledReturnTypeCount;

newFunctionBody->profiledLdElemCount = this->profiledLdElemCount;

newFunctionBody->profiledStElemCount = this->profiledStElemCount;

newFunctionBody->profiledSlotCount = this->profiledSlotCount;

newFunctionBody->flags = this->flags;

newFunctionBody->m\_isFuncRegistered = this->m\_isFuncRegistered;

newFunctionBody->m\_isFuncRegisteredToDiag = this->m\_isFuncRegisteredToDiag;

newFunctionBody->m\_hasBailoutInstrInJittedCode = this->m\_hasBailoutInstrInJittedCode;

newFunctionBody->m\_depth = this->m\_depth;

newFunctionBody->inlineDepth = 0;

newFunctionBody->recentlyBailedOutOfJittedLoopBody = false;

newFunctionBody->m\_pendingLoopHeaderRelease = this->m\_pendingLoopHeaderRelease;

newFunctionBody->m\_envDepth = this->m\_envDepth;

if (this->m\_constTable != nullptr)

{

this->CloneConstantTable(newFunctionBody);

}

newFunctionBody->cacheIdToPropertyIdMap = this->cacheIdToPropertyIdMap;

newFunctionBody->referencedPropertyIdMap = this->referencedPropertyIdMap;

newFunctionBody->propertyIdsForScopeSlotArray = this->propertyIdsForScopeSlotArray;

newFunctionBody->propertyIdOnRegSlotsContainer = this->propertyIdOnRegSlotsContainer;

newFunctionBody->scopeSlotArraySize = this->scopeSlotArraySize;

if (this->byteCodeBlock == nullptr)

{

newFunctionBody->SetDeferredParsingEntryPoint();

}

else

{

newFunctionBody->byteCodeBlock = this->byteCodeBlock->Clone(this->m\_scriptContext->GetRecycler());

newFunctionBody->isByteCodeDebugMode = this->isByteCodeDebugMode;

newFunctionBody->m\_byteCodeCount = this->m\_byteCodeCount;

newFunctionBody->m\_byteCodeWithoutLDACount = this->m\_byteCodeWithoutLDACount;

newFunctionBody->m\_byteCodeInLoopCount = this->m\_byteCodeInLoopCount;

#ifdef PERF\_COUNTERS

DWORD byteCodeSize = this->byteCodeBlock->GetLength();

#endif

if (this->auxBlock)

{

newFunctionBody->auxBlock = this->auxBlock->Clone(this->m\_scriptContext->GetRecycler());

#ifdef PERF\_COUNTERS

byteCodeSize += this->auxBlock->GetLength();

#endif

}

if (this->auxContextBlock)

{

newFunctionBody->auxContextBlock = this->auxContextBlock->Clone(scriptContext->GetRecycler(), scriptContext);

#ifdef PERF\_COUNTERS

byteCodeSize += this->auxContextBlock->GetLength();

#endif

}

if (this->GetProbeBackingBlock())

{

newFunctionBody->SetProbeBackingBlock(this->GetProbeBackingBlock()->Clone(scriptContext->GetRecycler()));

newFunctionBody->m\_sourceInfo.m\_probeCount = m\_sourceInfo.m\_probeCount;

}

#ifdef PERF\_COUNTERS

PERF\_COUNTER\_ADD(Code, DynamicByteCodeSize, byteCodeSize);

PERF\_COUNTER\_ADD(Code, TotalByteCodeSize, byteCodeSize);

#endif

newFunctionBody->SetFrameDisplayRegister(this->GetFrameDisplayRegister());

newFunctionBody->SetObjectRegister(this->GetObjectRegister());

StatementMapList \* pStatementMaps = this->GetStatementMaps();

if (pStatementMaps != nullptr)

{

Recycler\* recycler = newFunctionBody->GetScriptContext()->GetRecycler();

StatementMapList \* newStatementMaps = RecyclerNew(recycler, StatementMapList, recycler);

newFunctionBody->pStatementMaps = newStatementMaps;

pStatementMaps->Map([recycler, newStatementMaps](int index, FunctionBody::StatementMap\* oldStatementMap)

{

FunctionBody::StatementMap\* newStatementMap = StatementMap::New(recycler);

\*newStatementMap = \*oldStatementMap;

newStatementMaps->Add(newStatementMap);

});

}

if (this->m\_sourceInfo.pSpanSequence != nullptr)

{

// Span sequence is heap allocated

newFunctionBody->m\_sourceInfo.pSpanSequence = this->m\_sourceInfo.pSpanSequence->Clone();

}

Assert(newFunctionBody->GetDirectEntryPoint(newFunctionBody->GetDefaultEntryPointInfo()) == scriptContext->CurrentThunk);

Assert(newFunctionBody->IsInterpreterThunk());

}

// Create a new inline cache

newFunctionBody->inlineCacheCount = this->inlineCacheCount;

newFunctionBody->rootObjectLoadInlineCacheStart = this->rootObjectLoadInlineCacheStart;

newFunctionBody->rootObjectStoreInlineCacheStart = this->rootObjectStoreInlineCacheStart;

newFunctionBody->isInstInlineCacheCount = this->isInstInlineCacheCount;

newFunctionBody->referencedPropertyIdCount = this->referencedPropertyIdCount;

newFunctionBody->AllocateInlineCache();

newFunctionBody->objLiteralCount = this->objLiteralCount;

newFunctionBody->AllocateObjectLiteralTypeArray();

newFunctionBody->simpleJitEntryPointInfo = nullptr;

newFunctionBody->loopInterpreterLimit = loopInterpreterLimit;

newFunctionBody->ReinitializeExecutionModeAndLimits();

// Clone literal regexes

newFunctionBody->literalRegexCount = this->literalRegexCount;

newFunctionBody->AllocateLiteralRegexArray();

for(uint i = 0; i < this->literalRegexCount; ++i)

{

const auto literalRegex = this->literalRegexes[i];

if(!literalRegex)

{

Assert(!newFunctionBody->GetLiteralRegex(i));

continue;

}

const auto source = literalRegex->GetSource();

newFunctionBody->SetLiteralRegex(

i,

RegexHelper::CompileDynamic(

scriptContext,

source.GetBuffer(),

source.GetLength(),

literalRegex->GetFlags(),

true));

}

if (this->DoJITLoopBody())

{

newFunctionBody->AllocateLoopHeaders();

for (uint i = 0; i < this->GetLoopCount(); i++)

{

newFunctionBody->GetLoopHeader(i)->startOffset = GetLoopHeader(i)->startOffset;

newFunctionBody->GetLoopHeader(i)->endOffset = GetLoopHeader(i)->endOffset;

}

}

newFunctionBody->serializationIndex = this->serializationIndex;

newFunctionBody->m\_isFromNativeCodeModule = this->m\_isFromNativeCodeModule;

}

FunctionBody \*

FunctionBody::Clone(ScriptContext \* scriptContext, uint sourceIndex)

{

#if ENABLE\_NATIVE\_CODEGEN && defined(ENABLE\_PREJIT)

bool isNested = sourceIndex != Constants::InvalidSourceIndex;

#endif

Utf8SourceInfo\* sourceInfo = nullptr;

if(sourceIndex == Constants::InvalidSourceIndex)

{

// If we're copying a source info across script contexts, we need

// to create a copy of the Utf8SourceInfo (just the structure, not the source code itself)

// because a Utf8SourceInfo must reference only function bodies created within that script

// context

Utf8SourceInfo\* oldSourceInfo = GetUtf8SourceInfo();

SRCINFO\* srcInfo = GetHostSrcInfo()->Clone(scriptContext);

sourceInfo = scriptContext->CloneSourceCrossContext(oldSourceInfo, srcInfo);

sourceIndex = scriptContext->SaveSourceNoCopy(sourceInfo, oldSourceInfo->GetCchLength(), oldSourceInfo->GetIsCesu8());

}

else

{

sourceInfo = scriptContext->GetSource(sourceIndex);

}

FunctionBody \* newFunctionBody = FunctionBody::NewFromRecycler(scriptContext, this->GetDisplayName(), this->GetDisplayNameLength(),

this->GetShortDisplayNameOffset(), this->m\_nestedCount, sourceInfo, this->m\_functionNumber, this->m\_uScriptId,

this->GetLocalFunctionId(), this->m\_boundPropertyRecords,

this->GetAttributes()

#ifdef PERF\_COUNTERS

, false

#endif

);

if (this->m\_scopeInfo != nullptr)

{

newFunctionBody->SetScopeInfo(m\_scopeInfo->CloneFor(newFunctionBody));

}

newFunctionBody->CloneSourceInfo(scriptContext, (\*this), this->m\_scriptContext, sourceIndex);

CloneByteCodeInto(scriptContext, newFunctionBody, sourceIndex);

#if DBG

newFunctionBody->m\_iProfileSession = this->m\_iProfileSession;

newFunctionBody->deferredParseNextFunctionId = this->deferredParseNextFunctionId;

#endif

#if ENABLE\_PROFILE\_INFO

if (this->HasDynamicProfileInfo())

{

newFunctionBody->EnsureDynamicProfileInfo();

}

#endif

newFunctionBody->byteCodeCache = this->byteCodeCache;

#if ENABLE\_NATIVE\_CODEGEN

if (newFunctionBody->GetByteCode() && (IsIntermediateCodeGenThunk(this->GetOriginalEntryPoint())

|| IsNativeOriginalEntryPoint()))

{

#ifdef ENABLE\_PREJIT

if (Js::Configuration::Global.flags.Prejit)

{

if (!isNested)

{

GenerateAllFunctions(scriptContext->GetNativeCodeGenerator(), newFunctionBody);

}

}

else

#endif

{

GenerateFunction(scriptContext->GetNativeCodeGenerator(), newFunctionBody);

}

}

#endif

return newFunctionBody;

}

void FunctionBody::SetStackNestedFuncParent(FunctionBody \* parentFunctionBody)

{

Assert(this->stackNestedFuncParent == nullptr);

Assert(CanDoStackNestedFunc());

Assert(parentFunctionBody->DoStackNestedFunc());

this->stackNestedFuncParent = this->GetScriptContext()->GetRecycler()->CreateWeakReferenceHandle(parentFunctionBody);

}

FunctionBody \* FunctionBody::GetStackNestedFuncParent()

{

Assert(this->stackNestedFuncParent);

return this->stackNestedFuncParent->Get();

}

FunctionBody \* FunctionBody::GetAndClearStackNestedFuncParent()

{

if (this->stackNestedFuncParent)

{

FunctionBody \* parentFunctionBody = GetStackNestedFuncParent();

ClearStackNestedFuncParent();

return parentFunctionBody;

}

return nullptr;

}

void FunctionBody::ClearStackNestedFuncParent()

{

this->stackNestedFuncParent = nullptr;

}

ParseableFunctionInfo\* ParseableFunctionInfo::CopyFunctionInfoInto(ScriptContext \*scriptContext, Js::ParseableFunctionInfo\* newFunctionInfo, uint sourceIndex)

{

newFunctionInfo->m\_inParamCount = this->m\_inParamCount;

newFunctionInfo->m\_grfscr = this->m\_grfscr;

newFunctionInfo->m\_isDeclaration = this->m\_isDeclaration;

newFunctionInfo->m\_hasImplicitArgIns = this->m\_hasImplicitArgIns;

newFunctionInfo->m\_isAccessor = this->m\_isAccessor;

newFunctionInfo->m\_isGlobalFunc = this->m\_isGlobalFunc;

newFunctionInfo->m\_dontInline = this->m\_dontInline;

newFunctionInfo->m\_isTopLevel = this->m\_isTopLevel;

newFunctionInfo->m\_isPublicLibraryCode = this->m\_isPublicLibraryCode;

newFunctionInfo->scopeSlotArraySize = this->scopeSlotArraySize;

for (uint index = 0; index < this->m\_nestedCount; index++)

{

FunctionProxy\* proxy = this->GetNestedFunc(index);

if (proxy)

{

// Deserialize the proxy here if we have to

ParseableFunctionInfo\* body = proxy->EnsureDeserialized();

FunctionProxy\* newBody;

if (body->IsDeferredParseFunction())

{

newBody = body->Clone(scriptContext, sourceIndex);

}

else

{

newBody = body->GetFunctionBody()->Clone(scriptContext, sourceIndex);

}

// 0u is an empty value for the bit-mask 'flags', when initially parsing this is used to track defer-parse functions.

newFunctionInfo->SetNestedFunc(newBody, index, 0u);

}

else

{

// 0u is an empty value for the bit-mask 'flags', when initially parsing this is used to track defer-parse functions.

newFunctionInfo->SetNestedFunc(nullptr, index, 0u);

}

}

return newFunctionInfo;

}

ParseableFunctionInfo\* ParseableFunctionInfo::Clone(ScriptContext \*scriptContext, uint sourceIndex)

{

Utf8SourceInfo\* sourceInfo = nullptr;

if(sourceIndex == Constants::InvalidSourceIndex)

{

// If we're copying a source info across script contexts, we need

// to create a copy of the Utf8SourceInfo (just the structure, not the source code itself)

// because a Utf8SourceInfo must reference only function bodies created within that script

// context

Utf8SourceInfo\* oldSourceInfo = GetUtf8SourceInfo();

SRCINFO\* srcInfo = GetHostSrcInfo()->Clone(scriptContext);

sourceInfo = scriptContext->CloneSourceCrossContext(oldSourceInfo, srcInfo);

sourceIndex = scriptContext->SaveSourceNoCopy(sourceInfo, oldSourceInfo->GetCchLength(), oldSourceInfo->GetIsCesu8());

}

else

{

sourceInfo = scriptContext->GetSource(sourceIndex);

}

ParseableFunctionInfo\* newFunctionInfo = ParseableFunctionInfo::New(scriptContext, this->m\_nestedCount, this->GetLocalFunctionId(), sourceInfo, this->GetDisplayName(), this->GetDisplayNameLength(),

this->GetShortDisplayNameOffset(), this->m\_boundPropertyRecords, this->GetAttributes());

if (this->m\_scopeInfo != nullptr)

{

newFunctionInfo->SetScopeInfo(m\_scopeInfo->CloneFor(newFunctionInfo));

}

newFunctionInfo->CloneSourceInfo(scriptContext, (\*this), this->m\_scriptContext, sourceIndex);

CopyFunctionInfoInto(scriptContext, newFunctionInfo, sourceIndex);

#if DBG

newFunctionInfo->deferredParseNextFunctionId = this->deferredParseNextFunctionId;

#endif

return newFunctionInfo;

}

void FunctionBody::CreateCacheIdToPropertyIdMap(uint rootObjectLoadInlineCacheStart, uint rootObjectLoadMethodInlineCacheStart,

uint rootObjectStoreInlineCacheStart,

uint totalFieldAccessInlineCacheCount, uint isInstInlineCacheCount)

{

Assert(this->rootObjectLoadInlineCacheStart == 0);

Assert(this->rootObjectLoadMethodInlineCacheStart == 0);

Assert(this->rootObjectStoreInlineCacheStart == 0);

Assert(this->inlineCacheCount == 0);

Assert(this->isInstInlineCacheCount == 0);

this->rootObjectLoadInlineCacheStart = rootObjectLoadInlineCacheStart;

this->rootObjectLoadMethodInlineCacheStart = rootObjectLoadMethodInlineCacheStart;

this->rootObjectStoreInlineCacheStart = rootObjectStoreInlineCacheStart;

this->inlineCacheCount = totalFieldAccessInlineCacheCount;

this->isInstInlineCacheCount = isInstInlineCacheCount;

this->CreateCacheIdToPropertyIdMap();

}

void FunctionBody::CreateCacheIdToPropertyIdMap()

{

Assert(this->cacheIdToPropertyIdMap == nullptr);

Assert(this->inlineCaches == nullptr);

uint count = this->GetInlineCacheCount() ;

if (count!= 0)

{

this->cacheIdToPropertyIdMap =

RecyclerNewArrayLeaf(this->m\_scriptContext->GetRecycler(), PropertyId, count);

#if DBG

for (uint i = 0; i < count; i++)

{

this->cacheIdToPropertyIdMap[i] = Js::Constants::NoProperty;

}

#endif

}

}

#if DBG

void FunctionBody::VerifyCacheIdToPropertyIdMap()

{

uint count = this->GetInlineCacheCount();

for (uint i = 0; i < count; i++)

{

Assert(this->cacheIdToPropertyIdMap[i] != Js::Constants::NoProperty);

}

}

#endif

void FunctionBody::SetPropertyIdForCacheId(uint cacheId, PropertyId propertyId)

{

Assert(this->cacheIdToPropertyIdMap != nullptr);

Assert(cacheId < this->GetInlineCacheCount());

Assert(this->cacheIdToPropertyIdMap[cacheId] == Js::Constants::NoProperty);

this->cacheIdToPropertyIdMap[cacheId] = propertyId;

}

void FunctionBody::CreateReferencedPropertyIdMap(uint referencedPropertyIdCount)

{

this->referencedPropertyIdCount = referencedPropertyIdCount;

this->CreateReferencedPropertyIdMap();

}

void FunctionBody::CreateReferencedPropertyIdMap()

{

Assert(this->referencedPropertyIdMap == nullptr);

uint count = this->GetReferencedPropertyIdCount();

if (count!= 0)

{

this->referencedPropertyIdMap =

RecyclerNewArrayLeaf(this->m\_scriptContext->GetRecycler(), PropertyId, count);

#if DBG

for (uint i = 0; i < count; i++)

{

this->referencedPropertyIdMap[i] = Js::Constants::NoProperty;

}

#endif

}

}

#if DBG

void FunctionBody::VerifyReferencedPropertyIdMap()

{

uint count = this->GetReferencedPropertyIdCount();

for (uint i = 0; i < count; i++)

{

Assert(this->referencedPropertyIdMap[i] != Js::Constants::NoProperty);

}

}

#endif

PropertyId FunctionBody::GetReferencedPropertyId(uint index)

{

if (index < (uint)TotalNumberOfBuiltInProperties)

{

return index;

}

uint mapIndex = index - TotalNumberOfBuiltInProperties;

return GetReferencedPropertyIdWithMapIndex(mapIndex);

}

PropertyId FunctionBody::GetReferencedPropertyIdWithMapIndex(uint mapIndex)

{

Assert(this->referencedPropertyIdMap);

Assert(mapIndex < this->GetReferencedPropertyIdCount());

return this->referencedPropertyIdMap[mapIndex];

}

void FunctionBody::SetReferencedPropertyIdWithMapIndex(uint mapIndex, PropertyId propertyId)

{

Assert(propertyId >= TotalNumberOfBuiltInProperties);

Assert(mapIndex < this->GetReferencedPropertyIdCount());

Assert(this->referencedPropertyIdMap != nullptr);

Assert(this->referencedPropertyIdMap[mapIndex] == Js::Constants::NoProperty);

this->referencedPropertyIdMap[mapIndex] = propertyId;

}

void FunctionBody::CreateConstantTable()

{

Assert(this->m\_constTable == nullptr);

Assert(m\_constCount > FirstRegSlot);

this->m\_constTable = RecyclerNewArrayZ(this->m\_scriptContext->GetRecycler(), Var, m\_constCount);

// Initialize with the root object, which will always be recorded here.

Js::RootObjectBase \* rootObject = this->LoadRootObject();

if (rootObject)

{

this->RecordConstant(RootObjectRegSlot, rootObject);

}

else

{

Assert(false);

this->RecordConstant(RootObjectRegSlot, this->m\_scriptContext->GetLibrary()->GetUndefined());

}

}

void FunctionBody::RecordConstant(RegSlot location, Var var)

{

Assert(location < m\_constCount);

Assert(this->m\_constTable);

Assert(var != nullptr);

Assert(this->m\_constTable[location - FunctionBody::FirstRegSlot] == nullptr);

this->m\_constTable[location - FunctionBody::FirstRegSlot] = var;

}

void FunctionBody::RecordNullObject(RegSlot location)

{

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

Var nullObject = JavascriptOperators::OP\_LdNull(scriptContext);

this->RecordConstant(location, nullObject);

}

void FunctionBody::RecordUndefinedObject(RegSlot location)

{

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

Var undefObject = JavascriptOperators::OP\_LdUndef(scriptContext);

this->RecordConstant(location, undefObject);

}

void FunctionBody::RecordTrueObject(RegSlot location)

{

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

Var trueObject = JavascriptBoolean::OP\_LdTrue(scriptContext);

this->RecordConstant(location, trueObject);

}

void FunctionBody::RecordFalseObject(RegSlot location)

{

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

Var falseObject = JavascriptBoolean::OP\_LdFalse(scriptContext);

this->RecordConstant(location, falseObject);

}

void FunctionBody::RecordIntConstant(RegSlot location, unsigned int val)

{

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

Var intConst = JavascriptNumber::ToVar((int32)val, scriptContext);

this->RecordConstant(location, intConst);

}

void FunctionBody::RecordStrConstant(RegSlot location, LPCOLESTR psz, ulong cch)

{

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

PropertyRecord const \* propertyRecord;

scriptContext->FindPropertyRecord(psz, cch, &propertyRecord);

Var str;

if (propertyRecord == nullptr)

{

str = JavascriptString::NewCopyBuffer(psz, cch, scriptContext);

}

else

{

// If a particular string constant already has a propertyId, just create a property string for it

// as it might be likely that it is used for a property lookup

str = scriptContext->GetPropertyString(propertyRecord->GetPropertyId());

}

this->RecordConstant(location, str);

}

void FunctionBody::RecordFloatConstant(RegSlot location, double d)

{

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

Var floatConst = JavascriptNumber::ToVarIntCheck(d, scriptContext);

this->RecordConstant(location, floatConst);

}

void FunctionBody::RecordNullDisplayConstant(RegSlot location)

{

this->RecordConstant(location, (Js::Var)&Js::NullFrameDisplay);

}

void FunctionBody::RecordStrictNullDisplayConstant(RegSlot location)

{

this->RecordConstant(location, (Js::Var)&Js::StrictNullFrameDisplay);

}

void FunctionBody::InitConstantSlots(Var \*dstSlots)

{

// Initialize the given slots from the constant table.

Assert(m\_constCount > FunctionBody::FirstRegSlot);

js\_memcpy\_s(dstSlots, (m\_constCount - FunctionBody::FirstRegSlot) \* sizeof(Var), this->m\_constTable, (m\_constCount - FunctionBody::FirstRegSlot) \* sizeof(Var));

}

Var FunctionBody::GetConstantVar(RegSlot location)

{

Assert(this->m\_constTable);

Assert(location < m\_constCount);

Assert(location != 0);

return this->m\_constTable[location - FunctionBody::FirstRegSlot];

}

void FunctionBody::CloneConstantTable(FunctionBody \*newFunc)

{

// Creating the constant table initializes the root object.

newFunc->CreateConstantTable();

// Start walking the slots after the root object.

for (RegSlot reg = FunctionBody::RootObjectRegSlot + 1; reg < m\_constCount; reg++)

{

Var oldVar = this->GetConstantVar(reg);

Assert(oldVar != nullptr);

if (TaggedInt::Is(oldVar))

{

newFunc->RecordIntConstant(reg, TaggedInt::ToInt32(oldVar));

}

else if (oldVar == &Js::NullFrameDisplay)

{

newFunc->RecordNullDisplayConstant(reg);

}

else if (oldVar == &Js::StrictNullFrameDisplay)

{

newFunc->RecordStrictNullDisplayConstant(reg);

}

else

{

switch (JavascriptOperators::GetTypeId(oldVar))

{

case Js::TypeIds\_Undefined:

newFunc->RecordUndefinedObject(reg);

break;

case Js::TypeIds\_Null:

newFunc->RecordNullObject(reg);

break;

case Js::TypeIds\_Number:

newFunc->RecordFloatConstant(reg, JavascriptNumber::GetValue(oldVar));

break;

case Js::TypeIds\_String:

{

JavascriptString \*str = JavascriptString::FromVar(oldVar);

newFunc->RecordStrConstant(reg, str->GetSz(), str->GetLength());

break;

}

case Js::TypeIds\_ES5Array:

newFunc->RecordConstant(reg, oldVar);

break;

case Js::TypeIds\_Boolean:

if (Js::JavascriptBoolean::FromVar(oldVar)->GetValue())

{

newFunc->RecordTrueObject(reg);

}

else

{

newFunc->RecordFalseObject(reg);

}

break;

default:

AssertMsg(UNREACHED, "Unexpected object type in CloneConstantTable");

break;

}

}

}

}

#if DBG\_DUMP

void FunctionBody::Dump()

{

Js::ByteCodeDumper::Dump(this);

}

void FunctionBody::DumpScopes()

{

if(this->GetScopeObjectChain())

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"%s (%s) :\n", this->GetDisplayName(), this->GetDebugNumberSet(debugStringBuffer));

this->GetScopeObjectChain()->pScopeChain->Map( [=] (uint index, DebuggerScope\* scope )

{

scope->Dump();

});

}

}

#if ENABLE\_NATIVE\_CODEGEN

void EntryPointInfo::DumpNativeOffsetMaps()

{

// Native Offsets

if (this->nativeOffsetMaps.Count() > 0)

{

Output::Print(L"Native Map: baseAddr: 0x%0Ix, size: 0x%0Ix\nstatementId, offset range, address range\n",

this->GetNativeAddress(),

this->GetCodeSize());

int count = this->nativeOffsetMaps.Count();

for(int i = 0; i < count; i++)

{

const NativeOffsetMap\* map = &this->nativeOffsetMaps.Item(i);

Output::Print(L"S%4d, (%5d, %5d) (0x%012Ix, 0x%012Ix)\n", map->statementIndex,

map->nativeOffsetSpan.begin,

map->nativeOffsetSpan.end,

map->nativeOffsetSpan.begin + this->GetNativeAddress(),

map->nativeOffsetSpan.end + this->GetNativeAddress());

}

}

}

#endif

void FunctionBody::DumpStatementMaps()

{

// Source Map to ByteCode

StatementMapList \* pStatementMaps = this->GetStatementMaps();

if (pStatementMaps)

{

Output::Print(L"Statement Map:\nstatementId, SourceSpan, ByteCodeSpan\n");

int count = pStatementMaps->Count();

for(int i = 0; i < count; i++)

{

StatementMap\* map = pStatementMaps->Item(i);

Output::Print(L"S%4d, (C%5d, C%5d) (B%5d, B%5d) Inner=%d\n", i,

map->sourceSpan.begin,

map->sourceSpan.end,

map->byteCodeSpan.begin,

map->byteCodeSpan.end,

map->isSubexpression);

}

}

}

#if ENABLE\_NATIVE\_CODEGEN

void EntryPointInfo::DumpNativeThrowSpanSequence()

{

// Native Throw Map

if (this->nativeThrowSpanSequence)

{

Output::Print(L"Native Throw Map: baseAddr: 0x%0Ix, size: 0x%Ix\nstatementId, offset range, address range\n",

this->GetNativeAddress(),

this->GetCodeSize());

int count = this->nativeThrowSpanSequence->Count();

SmallSpanSequenceIter iter;

for (int i = 0; i < count; i++)

{

StatementData data;

if (this->nativeThrowSpanSequence->Item(i, iter, data))

{

Output::Print(L"S%4d, (%5d -----) (0x%012Ix --------)\n", data.sourceBegin, // statementIndex

data.bytecodeBegin, // nativeOffset

data.bytecodeBegin + this->GetNativeAddress());

}

}

}

}

#endif

void FunctionBody::PrintStatementSourceLine(uint statementIndex)

{

const uint startOffset = GetStatementStartOffset(statementIndex);

// startOffset should only be 0 if statementIndex is 0, otherwise it is EOF and we should skip printing anything

if (startOffset != 0 || statementIndex == 0)

{

PrintStatementSourceLineFromStartOffset(startOffset);

}

}

void FunctionBody::PrintStatementSourceLineFromStartOffset(uint cchStartOffset)

{

ULONG line;

LONG col;

LPCUTF8 source = GetStartOfDocument(L"FunctionBody::PrintStatementSourceLineFromStartOffset");

Utf8SourceInfo\* sourceInfo = this->GetUtf8SourceInfo();

Assert(sourceInfo != nullptr);

LPCUTF8 sourceInfoSrc = sourceInfo->GetSource(L"FunctionBody::PrintStatementSourceLineFromStartOffset");

if(!sourceInfoSrc)

{

Assert(sourceInfo->GetIsLibraryCode());

return;

}

if( source != sourceInfoSrc )

{

Output::Print(L"\nDETECTED MISMATCH:\n");

Output::Print(L"GetUtf8SourceInfo()->GetSource(): 0x%08X: %.\*s ...\n", sourceInfo, 16, sourceInfo);

Output::Print(L"GetStartOfDocument(): 0x%08X: %.\*s ...\n", source, 16, source);

AssertMsg(false, "Non-matching start of document");

}

GetLineCharOffsetFromStartChar(cchStartOffset, &line, &col, false /\*canAllocateLineCache\*/);

WORD color = 0;

if (Js::Configuration::Global.flags.DumpLineNoInColor)

{

color = Output::SetConsoleForeground(12);

}

Output::Print(L"\n\n Line %3d: ", line + 1);

// Need to match up cchStartOffset to appropriate cbStartOffset given function's cbStartOffset and cchStartOffset

size\_t i = utf8::CharacterIndexToByteIndex(source, sourceInfo->GetCbLength(), cchStartOffset, this->m\_cbStartOffset, this->m\_cchStartOffset);

size\_t lastOffset = StartOffset() + LengthInBytes();

for (;i < lastOffset && source[i] != '\n' && source[i] != '\r'; i++)

{

Output::Print(L"%C", source[i]);

}

Output::Print(L"\n");

Output::Print(L" Col %4d:%s^\n", col + 1, ((col+1)<10000) ? L" " : L"");

if (color != 0)

{

Output::SetConsoleForeground(color);

}

}

#endif // DBG\_DUMP

/\*\*

\* Get the source code offset for the given <statementIndex>.

\*/

uint FunctionBody::GetStatementStartOffset(const uint statementIndex)

{

uint startOffset = 0;

if (statementIndex != Js::Constants::NoStatementIndex)

{

const Js::FunctionBody::SourceInfo \* sourceInfo = &(this->m\_sourceInfo);

if (sourceInfo->pSpanSequence != nullptr)

{

Js::SmallSpanSequenceIter iter;

sourceInfo->pSpanSequence->Reset(iter);

Js::StatementData data;

sourceInfo->pSpanSequence->Item(statementIndex, iter, data);

startOffset = data.sourceBegin;

}

else

{

int index = statementIndex;

Js::FunctionBody::StatementMap \* statementMap = GetNextNonSubexpressionStatementMap(GetStatementMaps(), index);

startOffset = statementMap->sourceSpan.Begin();

}

}

return startOffset;

}

#ifdef IR\_VIEWER

/\* BEGIN potentially reusable code \*/

/\*

This code could be reused for locating source code in a debugger or to

retrieve the text of source statements.

Currently this code is used to retrieve the text of a source code statement

in the IR\_VIEWER feature.

\*/

/\*\*

\* Given a statement's starting offset in the source code, calculate the beginning and end of a statement,

\* as well as the line and column number where the statement appears.

\*

\* @param startOffset (input) The offset into the source code where this statement begins.

\* @param sourceBegin (output) The beginning of the statement in the source string.

\* @param sourceEnd (output) The end of the statement in the source string.

\* @param line (output) The line number where the statement appeared in the source.

\* @param col (output) The column number where the statement appeared in the source.

\*/

void FunctionBody::GetSourceLineFromStartOffset(const uint startOffset, LPCUTF8 \*sourceBegin, LPCUTF8 \*sourceEnd,

ULONG \* line, LONG \* col)

{

//

// get source info

//

LPCUTF8 source = GetStartOfDocument(L"IR Viewer FunctionBody::GetSourceLineFromStartOffset");

Utf8SourceInfo\* sourceInfo = this->GetUtf8SourceInfo();

Assert(sourceInfo != nullptr);

LPCUTF8 sourceInfoSrc = sourceInfo->GetSource(L"IR Viewer FunctionBody::GetSourceLineFromStartOffset");

if (!sourceInfoSrc)

{

Assert(sourceInfo->GetIsLibraryCode());

return;

}

if (source != sourceInfoSrc)

{

Output::Print(L"\nDETECTED MISMATCH:\n");

Output::Print(L"GetUtf8SourceInfo()->GetSource(): 0x%08X: %.\*s ...\n", sourceInfo, 16, sourceInfo);

Output::Print(L"GetStartOfDocument(): 0x%08X: %.\*s ...\n", source, 16, source);

AssertMsg(false, "Non-matching start of document");

}

//

// calculate source line info

//

size\_t cbStartOffset = utf8::CharacterIndexToByteIndex(source, sourceInfo->GetCbLength(), (const charcount\_t)startOffset, (size\_t)this->m\_cbStartOffset, (charcount\_t)this->m\_cchStartOffset);

GetLineCharOffsetFromStartChar(startOffset, line, col);

size\_t lastOffset = StartOffset() + LengthInBytes();

size\_t i = 0;

for (i = cbStartOffset; i < lastOffset && source[i] != '\n' && source[i] != '\r'; i++)

{

// do nothing; scan until end of statement

}

size\_t cbEndOffset = i;

//

// return

//

\*sourceBegin = &source[cbStartOffset];

\*sourceEnd = &source[cbEndOffset];

}

/\*\*

\* Given a statement index and output parameters, calculate the beginning and end of a statement,

\* as well as the line and column number where the statement appears.

\*

\* @param statementIndex (input) The statement's index (as used by the StatementBoundary pragma).

\* @param sourceBegin (output) The beginning of the statement in the source string.

\* @param sourceEnd (output) The end of the statement in the source string.

\* @param line (output) The line number where the statement appeared in the source.

\* @param col (output) The column number where the statement appeared in the source.

\*/

void FunctionBody::GetStatementSourceInfo(const uint statementIndex, LPCUTF8 \*sourceBegin, LPCUTF8 \*sourceEnd,

ULONG \* line, LONG \* col)

{

const size\_t startOffset = GetStatementStartOffset(statementIndex);

// startOffset should only be 0 if statementIndex is 0, otherwise it is EOF and we should return empty string

if (startOffset != 0 || statementIndex == 0)

{

GetSourceLineFromStartOffset(startOffset, sourceBegin, sourceEnd, line, col);

}

else

{

\*sourceBegin = nullptr;

\*sourceEnd = nullptr;

\*line = 0;

\*col = 0;

return;

}

}

/\* END potentially reusable code \*/

#endif /\* IR\_VIEWER \*/

#ifdef IR\_VIEWER

Js::DynamicObject \* FunctionBody::GetIRDumpBaseObject()

{

if (!this->m\_irDumpBaseObject)

{

this->m\_irDumpBaseObject = this->m\_scriptContext->GetLibrary()->CreateObject();

}

return this->m\_irDumpBaseObject;

}

#endif /\* IR\_VIEWER \*/

#ifdef VTUNE\_PROFILING

#ifdef CDECL

#define ORIGINAL\_CDECL CDECL

#undef CDECL

#endif

// Not enabled in ChakraCore

#include "jitProfiling.h"

#ifdef ORIGINAL\_CDECL

#undef CDECL

#endif

#define CDECL ORIGINAL\_CDECL

int EntryPointInfo::GetNativeOffsetMapCount() const

{

return this->nativeOffsetMaps.Count();

}

uint EntryPointInfo::PopulateLineInfo(void\* pInfo, FunctionBody\* body)

{

LineNumberInfo\* pLineInfo = (LineNumberInfo\*)pInfo;

ULONG functionLineNumber = body->GetLineNumber();

pLineInfo[0].Offset = 0;

pLineInfo[0].LineNumber = functionLineNumber;

int lineNumber = 0;

int j = 1; // start with 1 since offset 0 has already been populated with function line number

int count = this->nativeOffsetMaps.Count();

for(int i = 0; i < count; i++)

{

const NativeOffsetMap\* map = &this->nativeOffsetMaps.Item(i);

uint32 statementIndex = map->statementIndex;

if (statementIndex == 0)

{

// statementIndex is 0, first line in the function, populate with function line number

pLineInfo[j].Offset = map->nativeOffsetSpan.begin;

pLineInfo[j].LineNumber = functionLineNumber;

j++;

}

lineNumber = body->GetSourceLineNumber(statementIndex);

if (lineNumber != 0)

{

pLineInfo[j].Offset = map->nativeOffsetSpan.end;

pLineInfo[j].LineNumber = lineNumber;

j++;

}

}

return j;

}

ULONG FunctionBody::GetSourceLineNumber(uint statementIndex)

{

ULONG line = 0;

if (statementIndex != Js::Constants::NoStatementIndex)

{

uint startOffset = GetStartOffset(statementIndex);

if (startOffset != 0 || statementIndex == 0)

{

GetLineCharOffsetFromStartChar(startOffset, &line, nullptr, false /\*canAllocateLineCache\*/);

line = line + 1;

}

}

return line;

}

uint FunctionBody::GetStartOffset(uint statementIndex) const

{

uint startOffset = 0;

const Js::FunctionBody::SourceInfo \* sourceInfo = &this->m\_sourceInfo;

if (sourceInfo->pSpanSequence != nullptr)

{

Js::SmallSpanSequenceIter iter;

sourceInfo->pSpanSequence->Reset(iter);

Js::StatementData data;

sourceInfo->pSpanSequence->Item(statementIndex, iter, data);

startOffset = data.sourceBegin;

}

else

{

int index = statementIndex;

Js::FunctionBody::StatementMap \* statementMap = GetNextNonSubexpressionStatementMap(GetStatementMaps(), index);

startOffset = statementMap->sourceSpan.Begin();

}

return startOffset;

}

#endif

void FunctionBody::SetIsNonUserCode(bool set)

{

// Mark current function as a non-user code, so that we can distinguish cases where exceptions are

// caught in non-user code (see ProbeContainer::HasAllowedForException).

SetFlags(set, Flags\_NonUserCode);

// Propagate setting for all functions in this scope (nested).

for (uint uIndex = 0; uIndex < this->m\_nestedCount; uIndex++)

{

Js::FunctionBody \* pBody = this->GetNestedFunc(uIndex)->GetFunctionBody();

if (pBody != nullptr)

{

pBody->SetIsNonUserCode(set);

}

}

}

void FunctionBody::InsertSymbolToRegSlotList(JsUtil::CharacterBuffer<WCHAR> const& propName, RegSlot reg, RegSlot totalRegsCount)

{

if (totalRegsCount > 0)

{

PropertyId propertyId = GetOrAddPropertyIdTracked(propName);

InsertSymbolToRegSlotList(reg, propertyId, totalRegsCount);

}

}

void FunctionBody::InsertSymbolToRegSlotList(RegSlot reg, PropertyId propertyId, RegSlot totalRegsCount)

{

if (totalRegsCount > 0)

{

if (this->propertyIdOnRegSlotsContainer == nullptr)

{

this->propertyIdOnRegSlotsContainer = PropertyIdOnRegSlotsContainer::New(m\_scriptContext->GetRecycler());

}

if (this->propertyIdOnRegSlotsContainer->propertyIdsForRegSlots == nullptr)

{

this->propertyIdOnRegSlotsContainer->CreateRegSlotsArray(m\_scriptContext->GetRecycler(), totalRegsCount);

}

Assert(this->propertyIdOnRegSlotsContainer != nullptr);

this->propertyIdOnRegSlotsContainer->Insert(reg, propertyId);

}

}

void FunctionBody::SetPropertyIdsOfFormals(PropertyIdArray \* formalArgs)

{

Assert(formalArgs);

if (this->propertyIdOnRegSlotsContainer == nullptr)

{

this->propertyIdOnRegSlotsContainer = PropertyIdOnRegSlotsContainer::New(m\_scriptContext->GetRecycler());

}

this->propertyIdOnRegSlotsContainer->SetFormalArgs(formalArgs);

}

HRESULT FunctionBody::RegisterFunction(BOOL fChangeMode, BOOL fOnlyCurrent)

{

if (!this->IsFunctionParsed())

{

return S\_OK;

}

HRESULT hr = this->ReportFunctionCompiled();

if (FAILED(hr))

{

return hr;

}

if (fChangeMode)

{

this->SetEntryToProfileMode();

}

if (!fOnlyCurrent)

{

for (uint uIndex = 0; uIndex < this->m\_nestedCount; uIndex++)

{

Js::ParseableFunctionInfo \* pBody = this->GetNestedFunctionForExecution(uIndex);

if (pBody == nullptr || !pBody->IsFunctionParsed())

{

continue;

}

hr = pBody->GetFunctionBody()->RegisterFunction(fChangeMode);

if (FAILED(hr))

{

break;

}

}

}

return hr;

}

HRESULT FunctionBody::ReportScriptCompiled()

{

AssertMsg(m\_scriptContext != nullptr, "Script Context is null when reporting function information");

PROFILER\_SCRIPT\_TYPE type = IsDynamicScript() ? PROFILER\_SCRIPT\_TYPE\_DYNAMIC : PROFILER\_SCRIPT\_TYPE\_USER;

IDebugDocumentContext \*pDebugDocumentContext = nullptr;

this->m\_scriptContext->GetDocumentContext(this->m\_scriptContext, this, &pDebugDocumentContext);

HRESULT hr = m\_scriptContext->OnScriptCompiled((PROFILER\_TOKEN) this->GetUtf8SourceInfo()->GetSourceInfoId(), type, pDebugDocumentContext);

RELEASEPTR(pDebugDocumentContext);

return hr;

}

HRESULT FunctionBody::ReportFunctionCompiled()

{

// Some assumptions by Logger interface.

// to send NULL as a name in case the name is anonymous and hint is anonymous code.

const wchar\_t \*pwszName = GetExternalDisplayName();

IDebugDocumentContext \*pDebugDocumentContext = nullptr;

this->m\_scriptContext->GetDocumentContext(this->m\_scriptContext, this, &pDebugDocumentContext);

SetHasFunctionCompiledSent(true);

HRESULT hr = m\_scriptContext->OnFunctionCompiled(m\_functionNumber, (PROFILER\_TOKEN) this->GetUtf8SourceInfo()->GetSourceInfoId(), pwszName, nullptr, pDebugDocumentContext);

RELEASEPTR(pDebugDocumentContext);

#if DBG

if (m\_iProfileSession >= m\_scriptContext->GetProfileSession())

{

OUTPUT\_TRACE\_DEBUGONLY(Js::ScriptProfilerPhase, L"FunctionBody::ReportFunctionCompiled, Duplicate compile event (%d < %d) for FunctionNumber : %d\n",

m\_iProfileSession, m\_scriptContext->GetProfileSession(), m\_functionNumber);

}

AssertMsg(m\_iProfileSession < m\_scriptContext->GetProfileSession(), "Duplicate compile event sent");

m\_iProfileSession = m\_scriptContext->GetProfileSession();

#endif

return hr;

}

void FunctionBody::SetEntryToProfileMode()

{

#if ENABLE\_NATIVE\_CODEGEN

AssertMsg(this->m\_scriptContext->CurrentThunk == ProfileEntryThunk, "ScriptContext not in profile mode");

#if DBG

AssertMsg(m\_iProfileSession == m\_scriptContext->GetProfileSession(), "Changing mode to profile for function that didn't send compile event");

#endif

// This is always done when bg thread is paused hence we don't need any kind of thread-synchronization at this point.

// Change entry points to Profile Thunk

// If the entrypoint is CodeGenOnDemand or CodeGen - then we don't change the entry points

ProxyEntryPointInfo\* defaultEntryPointInfo = this->GetDefaultEntryPointInfo();

if (!IsIntermediateCodeGenThunk((JavascriptMethod) defaultEntryPointInfo->address)

&& defaultEntryPointInfo->address != DynamicProfileInfo::EnsureDynamicProfileInfoThunk)

{

if (this->originalEntryPoint == DefaultDeferredParsingThunk)

{

defaultEntryPointInfo->address = ProfileDeferredParsingThunk;

}

else if (this->originalEntryPoint == DefaultDeferredDeserializeThunk)

{

defaultEntryPointInfo->address = ProfileDeferredDeserializeThunk;

}

else

{

defaultEntryPointInfo->address = ProfileEntryThunk;

}

}

// Update old entry points on the deferred prototype type so that they match current defaultEntryPointInfo.

// to make sure that new JavascriptFunction instances use profile thunk.

if (this->deferredPrototypeType)

{

this->deferredPrototypeType->SetEntryPoint((JavascriptMethod)this->GetDefaultEntryPointInfo()->address);

this->deferredPrototypeType->SetEntryPointInfo(this->GetDefaultEntryPointInfo());

}

#if DBG

if (!this->HasValidEntryPoint())

{

OUTPUT\_TRACE\_DEBUGONLY(Js::ScriptProfilerPhase, L"FunctionBody::SetEntryToProfileMode, Assert due to HasValidEntryPoint(), directEntrypoint : 0x%0IX, originalentrypoint : 0x%0IX\n",

(JavascriptMethod)this->GetDefaultEntryPointInfo()->address, this->originalEntryPoint);

AssertMsg(false, "Not a valid EntryPoint");

}

#endif

#endif //ENABLE\_NATIVE\_CODEGEN

}

#if DBG

void FunctionBody::MustBeInDebugMode()

{

Assert(m\_scriptContext->IsInDebugMode());

Assert(IsByteCodeDebugMode());

Assert(m\_sourceInfo.pSpanSequence == nullptr);

Assert(pStatementMaps != nullptr);

}

#endif

void FunctionBody::CleanupToReparse()

{

// The current function is already compiled. In order to prep this function to ready for debug mode, most of the previous information need to be thrown away.

// Clean up the nested functions

for (uint i = 0; i < m\_nestedCount; i++)

{

FunctionProxy\* proxy = GetNestedFunc(i);

if (proxy && proxy->IsFunctionBody())

{

proxy->GetFunctionBody()->CleanupToReparse();

}

}

CleanupRecyclerData(/\* isShutdown \*/ false, true /\* capture entry point cleanup stack trace \*/);

this->entryPoints->ClearAndZero();

#if DYNAMIC\_INTERPRETER\_THUNK

if (m\_isAsmJsFunction && m\_dynamicInterpreterThunk)

{

m\_scriptContext->ReleaseDynamicAsmJsInterpreterThunk((BYTE\*)this->m\_dynamicInterpreterThunk, true);

this->m\_dynamicInterpreterThunk = nullptr;

}

#endif

// Store the originalEntryPoint to restore it back immediately.

JavascriptMethod originalEntryPoint = this->originalEntryPoint;

this->CreateNewDefaultEntryPoint();

this->originalEntryPoint = originalEntryPoint;

if (this->m\_defaultEntryPointInfo)

{

this->GetDefaultFunctionEntryPointInfo()->entryPointIndex = 0;

}

this->auxBlock = nullptr;

this->auxContextBlock = nullptr;

this->byteCodeBlock = nullptr;

this->loopHeaderArray = nullptr;

this->m\_constTable = nullptr;

this->m\_scopeInfo = nullptr;

this->m\_codeGenRuntimeData = nullptr;

this->m\_codeGenGetSetRuntimeData = nullptr;

this->cacheIdToPropertyIdMap = nullptr;

this->referencedPropertyIdMap = nullptr;

this->literalRegexes = nullptr;

this->propertyIdsForScopeSlotArray = nullptr;

this->propertyIdOnRegSlotsContainer = nullptr;

this->pStatementMaps = nullptr;

this->profiledLdElemCount = 0;

this->profiledStElemCount = 0;

this->profiledCallSiteCount = 0;

this->profiledArrayCallSiteCount = 0;

this->profiledDivOrRemCount = 0;

this->profiledSwitchCount = 0;

this->profiledReturnTypeCount = 0;

this->profiledSlotCount = 0;

this->loopCount = 0;

this->m\_envDepth = (uint16)-1;

this->m\_byteCodeCount = 0;

this->m\_byteCodeWithoutLDACount = 0;

this->m\_byteCodeInLoopCount = 0;

this->functionBailOutRecord = nullptr;

#if ENABLE\_PROFILE\_INFO

this->dynamicProfileInfo = nullptr;

#endif

this->hasExecutionDynamicProfileInfo = false;

this->m\_firstTmpReg = Constants::NoRegister;

this->m\_varCount = 0;

this->m\_constCount = 0;

this->localClosureRegister = Constants::NoRegister;

this->localFrameDisplayRegister = Constants::NoRegister;

this->envRegister = Constants::NoRegister;

this->thisRegisterForEventHandler = Constants::NoRegister;

this->firstInnerScopeRegister = Constants::NoRegister;

this->funcExprScopeRegister = Constants::NoRegister;

this->innerScopeCount = 0;

this->hasCachedScopePropIds = false;

this->ResetObjectLiteralTypes();

this->inlineCacheCount = 0;

this->rootObjectLoadInlineCacheStart = 0;

this->rootObjectLoadMethodInlineCacheStart = 0;

this->rootObjectStoreInlineCacheStart = 0;

this->isInstInlineCacheCount = 0;

this->m\_inlineCachesOnFunctionObject = false;

this->referencedPropertyIdCount = 0;

#if ENABLE\_PROFILE\_INFO

this->polymorphicCallSiteInfoHead = nullptr;

#endif

this->interpretedCount = 0;

this->m\_hasDoneAllNonLocalReferenced = false;

this->debuggerScopeIndex = 0;

this->m\_utf8SourceInfo->DeleteLineOffsetCache();

// Reset to default.

this->flags = Flags\_HasNoExplicitReturnValue;

ResetInParams();

this->m\_isAsmjsMode = false;

this->m\_isAsmJsFunction = false;

this->m\_isAsmJsScheduledForFullJIT = false;

this->m\_asmJsTotalLoopCount = 0;

recentlyBailedOutOfJittedLoopBody = false;

loopInterpreterLimit = CONFIG\_FLAG(LoopInterpretCount);

ReinitializeExecutionModeAndLimits();

Assert(this->m\_sourceInfo.m\_probeCount == 0);

this->m\_sourceInfo.m\_probeBackingBlock = nullptr;

#if DBG

// This could be non-zero if the function threw exception before. Reset it.

this->m\_DEBUG\_executionCount = 0;

#endif

if (this->m\_sourceInfo.pSpanSequence != nullptr)

{

HeapDelete(this->m\_sourceInfo.pSpanSequence);

this->m\_sourceInfo.pSpanSequence = nullptr;

}

if (this->m\_sourceInfo.m\_auxStatementData != nullptr)

{

// This must be consistent with how we allocate the data for this and inner structures.

// We are using recycler, thus it's enough just to set to NULL.

Assert(m\_scriptContext->GetRecycler()->IsValidObject(m\_sourceInfo.m\_auxStatementData));

m\_sourceInfo.m\_auxStatementData = nullptr;

}

}

void FunctionBody::SetEntryToDeferParseForDebugger()

{

ProxyEntryPointInfo\* defaultEntryPointInfo = this->GetDefaultEntryPointInfo();

if (defaultEntryPointInfo->address != DefaultDeferredParsingThunk && defaultEntryPointInfo->address != ProfileDeferredParsingThunk)

{

// Just change the thunk, the cleanup will be done once the function gets called.

if (this->m\_scriptContext->CurrentThunk == ProfileEntryThunk)

{

defaultEntryPointInfo->address = ProfileDeferredParsingThunk;

}

else

{

defaultEntryPointInfo->address = DefaultDeferredParsingThunk;

}

this->originalEntryPoint = DefaultDeferredParsingThunk;

// Abandon the shared type so a new function will get a new one

this->deferredPrototypeType = nullptr;

this->attributes = (FunctionInfo::Attributes) (this->attributes | FunctionInfo::Attributes::DeferredParse);

}

// Set other state back to before parse as well

this->SetStackNestedFunc(false);

this->stackNestedFuncParent = nullptr;

this->SetReparsed(true);

#if DBG

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

OUTPUT\_VERBOSE\_TRACE(Js::DebuggerPhase, L"Regenerate Due To Debug Mode: function %s (%s) from script context %p\n",

this->GetDisplayName(), this->GetDebugNumberSet(debugStringBuffer), m\_scriptContext);

#endif

}

//

// For library code all references to jitted entry points need to be removed

//

void FunctionBody::ResetEntryPoint()

{

if (this->entryPoints)

{

this->MapEntryPoints([] (int index, FunctionEntryPointInfo\* entryPoint)

{

if (nullptr != entryPoint)

{

// Finalize = Free up work item if it hasn't been released yet + entry point clean up

// isShutdown is false because cleanup is called only in the !isShutdown case

entryPoint->Finalize(/\*isShutdown\*/ false);

}

});

this->MapLoopHeaders([] (uint loopNumber, LoopHeader\* header)

{

header->MapEntryPoints([] (int index, LoopEntryPointInfo\* entryPoint)

{

entryPoint->Cleanup(/\*isShutdown\*/ false, true /\* capture cleanup stack \*/);

});

});

}

this->entryPoints->ClearAndZero();

this->CreateNewDefaultEntryPoint();

this->originalEntryPoint = DefaultEntryThunk;

m\_defaultEntryPointInfo->address = m\_scriptContext->CurrentThunk;

if (this->deferredPrototypeType)

{

// Update old entry points on the deferred prototype type,

// as they may point to old native code gen regions which age gone now.

this->deferredPrototypeType->SetEntryPoint((JavascriptMethod)this->GetDefaultEntryPointInfo()->address);

this->deferredPrototypeType->SetEntryPointInfo(this->GetDefaultEntryPointInfo());

}

ReinitializeExecutionModeAndLimits();

}

void FunctionBody::AddDeferParseAttribute()

{

this->attributes = (FunctionInfo::Attributes) (this->attributes | DeferredParse);

}

void FunctionBody::RemoveDeferParseAttribute()

{

this->attributes = (FunctionInfo::Attributes) (this->attributes & (~DeferredParse));

}

Js::DebuggerScope \* FunctionBody::GetDiagCatchScopeObjectAt(int byteCodeOffset)

{

if (GetScopeObjectChain())

{

for (int i = 0; i < GetScopeObjectChain()->pScopeChain->Count(); i++)

{

Js::DebuggerScope \*debuggerScope = GetScopeObjectChain()->pScopeChain->Item(i);

Assert(debuggerScope);

if (debuggerScope->IsCatchScope() && debuggerScope->IsOffsetInScope(byteCodeOffset))

{

return debuggerScope;

}

}

}

return nullptr;

}

ushort SmallSpanSequence::GetDiff(int current, int prev)

{

int diff = current - prev;

if ((diff) < SHRT\_MIN || (diff) >= SHRT\_MAX)

{

diff = SHRT\_MAX;

if (!this->pActualOffsetList)

{

this->pActualOffsetList = JsUtil::GrowingUint32HeapArray::Create(4);

}

this->pActualOffsetList->Add(current);

}

return (ushort)diff;

}

// Get Values of the beginning of the statement at particular index.

BOOL SmallSpanSequence::GetRangeAt(int index, SmallSpanSequenceIter &iter, int \* pCountOfMissed, StatementData & data)

{

Assert((uint32)index < pStatementBuffer->Count());

SmallSpan span(pStatementBuffer->ItemInBuffer((uint32)index));

int countOfMissed = 0;

if ((short)span.sourceBegin == SHRT\_MAX)

{

// Look in ActualOffset store

Assert(this->pActualOffsetList);

Assert(this->pActualOffsetList->Count() > 0);

Assert(this->pActualOffsetList->Count() > (uint32)iter.indexOfActualOffset);

data.sourceBegin = this->pActualOffsetList->ItemInBuffer((uint32)iter.indexOfActualOffset);

countOfMissed++;

}

else

{

data.sourceBegin = iter.accumulatedSourceBegin + (short)span.sourceBegin;

}

if (span.bytecodeBegin == SHRT\_MAX)

{

// Look in ActualOffset store

Assert(this->pActualOffsetList);

Assert(this->pActualOffsetList->Count() > 0);

Assert(this->pActualOffsetList->Count() > (uint32)(iter.indexOfActualOffset + countOfMissed));

data.bytecodeBegin = this->pActualOffsetList->ItemInBuffer((uint32)iter.indexOfActualOffset + countOfMissed);

countOfMissed++;

}

else

{

data.bytecodeBegin = iter.accumulatedBytecodeBegin + span.bytecodeBegin;

}

if (pCountOfMissed)

{

\*pCountOfMissed = countOfMissed;

}

return TRUE;

}

void SmallSpanSequence::Reset(SmallSpanSequenceIter &iter)

{

iter.accumulatedIndex = 0;

iter.accumulatedSourceBegin = baseValue;

iter.accumulatedBytecodeBegin = 0;

iter.indexOfActualOffset = 0;

}

BOOL SmallSpanSequence::GetMatchingStatementFromBytecode(int bytecode, SmallSpanSequenceIter &iter, StatementData & data)

{

if (Count() > 0 && bytecode >= 0)

{

// Support only in forward direction

if (bytecode < iter.accumulatedBytecodeBegin

|| iter.accumulatedIndex <= 0 || (uint32)iter.accumulatedIndex >= Count())

{

// re-initialize the accumulators

Reset(iter);

}

while ((uint32)iter.accumulatedIndex < Count())

{

int countOfMissed = 0;

if (!GetRangeAt(iter.accumulatedIndex, iter, &countOfMissed, data))

{

Assert(FALSE);

break;

}

if (data.bytecodeBegin >= bytecode)

{

if (data.bytecodeBegin > bytecode)

{

// Not exactly at the current bytecode, so it falls in between previous statement.

data.sourceBegin = iter.accumulatedSourceBegin;

data.bytecodeBegin = iter.accumulatedBytecodeBegin;

}

return TRUE;

}

// Look for the next

iter.accumulatedSourceBegin = data.sourceBegin;

iter.accumulatedBytecodeBegin = data.bytecodeBegin;

iter.accumulatedIndex++;

if (countOfMissed)

{

iter.indexOfActualOffset += countOfMissed;

}

}

if (iter.accumulatedIndex != -1)

{

// Give the last one.

Assert(data.bytecodeBegin < bytecode);

return TRUE;

}

}

// Failed to give the correct one, init to default

iter.accumulatedIndex = -1;

return FALSE;

}

BOOL SmallSpanSequence::Item(int index, SmallSpanSequenceIter &iter, StatementData & data)

{

if (!pStatementBuffer || (uint32)index >= pStatementBuffer->Count())

{

return FALSE;

}

if (iter.accumulatedIndex <= 0 || iter.accumulatedIndex > index)

{

Reset(iter);

}

while (iter.accumulatedIndex <= index)

{

Assert((uint32)iter.accumulatedIndex < pStatementBuffer->Count());

int countOfMissed = 0;

if (!GetRangeAt(iter.accumulatedIndex, iter, &countOfMissed, data))

{

Assert(FALSE);

break;

}

// We store the next index

iter.accumulatedSourceBegin = data.sourceBegin;

iter.accumulatedBytecodeBegin = data.bytecodeBegin;

iter.accumulatedIndex++;

if (countOfMissed)

{

iter.indexOfActualOffset += countOfMissed;

}

if ((iter.accumulatedIndex - 1) == index)

{

return TRUE;

}

}

return FALSE;

}

BOOL SmallSpanSequence::Seek(int index, StatementData & data)

{

// This method will not alter any state of the variables, so this will just do plain search

// from the beginning to look for that index.

SmallSpanSequenceIter iter;

Reset(iter);

return Item(index, iter, data);

}

SmallSpanSequence \* SmallSpanSequence::Clone()

{

SmallSpanSequence \*pNewSequence = HeapNew(SmallSpanSequence);

pNewSequence->baseValue = baseValue;

if (pStatementBuffer)

{

pNewSequence->pStatementBuffer = pStatementBuffer->Clone();

}

if (pActualOffsetList)

{

pNewSequence->pActualOffsetList = pActualOffsetList->Clone();

}

return pNewSequence;

}

PropertyIdOnRegSlotsContainer \* PropertyIdOnRegSlotsContainer::New(Recycler \* recycler)

{

return RecyclerNew(recycler, PropertyIdOnRegSlotsContainer);

}

PropertyIdOnRegSlotsContainer::PropertyIdOnRegSlotsContainer()

: propertyIdsForRegSlots(nullptr), length(0), propertyIdsForFormalArgs(nullptr)

{

}

void PropertyIdOnRegSlotsContainer::CreateRegSlotsArray(Recycler \* recycler, uint \_length)

{

Assert(propertyIdsForRegSlots == nullptr);

propertyIdsForRegSlots = RecyclerNewArrayLeafZ(recycler, PropertyId, \_length);

length = \_length;

}

void PropertyIdOnRegSlotsContainer::SetFormalArgs(PropertyIdArray \* formalArgs)

{

propertyIdsForFormalArgs = formalArgs;

}

//

// Helper methods for PropertyIdOnRegSlotsContainer

void PropertyIdOnRegSlotsContainer::Insert(RegSlot reg, PropertyId propId)

{

//

// Reg is being used as an index;

Assert(propertyIdsForRegSlots);

Assert(reg < length);

//

// the current reg is unaccounted for const reg count. while fetching calculate the actual regslot value.

Assert(propertyIdsForRegSlots[reg] == 0 || propertyIdsForRegSlots[reg] == propId);

propertyIdsForRegSlots[reg] = propId;

}

void PropertyIdOnRegSlotsContainer::FetchItemAt(uint index, FunctionBody \*pFuncBody, \_\_out PropertyId \*pPropId, \_\_out RegSlot \*pRegSlot)

{

Assert(index < length);

Assert(pPropId);

Assert(pRegSlot);

Assert(pFuncBody);

\*pPropId = propertyIdsForRegSlots[index];

\*pRegSlot = pFuncBody->MapRegSlot(index);

}

bool PropertyIdOnRegSlotsContainer::IsRegSlotFormal(RegSlot reg)

{

if (propertyIdsForFormalArgs != nullptr && reg < length)

{

PropertyId propId = propertyIdsForRegSlots[reg];

for (uint32 i = 0; i < propertyIdsForFormalArgs->count; i++)

{

if (propertyIdsForFormalArgs->elements[i] == propId)

{

return true;

}

}

}

return false;

}

ScopeType FrameDisplay::GetScopeType(void\* scope)

{

if(Js::ActivationObject::Is(scope))

{

return ScopeType\_ActivationObject;

}

if(Js::ScopeSlots::Is(scope))

{

return ScopeType\_SlotArray;

}

return ScopeType\_WithScope;

}

// DebuggerScope

// Get the sibling for the current debugger scope.

DebuggerScope \* DebuggerScope::GetSiblingScope(RegSlot location, FunctionBody \*functionBody)

{

bool isBlockSlotOrObject = scopeType == Js::DiagExtraScopesType::DiagBlockScopeInSlot || scopeType == Js::DiagExtraScopesType::DiagBlockScopeInObject;

bool isCatchSlotOrObject = scopeType == Js::DiagExtraScopesType::DiagCatchScopeInSlot || scopeType == Js::DiagExtraScopesType::DiagCatchScopeInObject;

// This is expected to be called only when the current scope is either slot or activation object.

Assert(isBlockSlotOrObject || isCatchSlotOrObject);

if (siblingScope == nullptr)

{

// If the sibling isn't there, attempt to retrieve it if we're reparsing or create it anew if this is the first parse.

siblingScope = functionBody->RecordStartScopeObject(isBlockSlotOrObject ? Js::DiagExtraScopesType::DiagBlockScopeDirect : Js::DiagExtraScopesType::DiagCatchScopeDirect, GetStart(), location);

}

return siblingScope;

}

// Adds a new property to be tracked in the debugger scope.

// location - The slot array index or register slot location of where the property is stored.

// propertyId - The property ID of the property.

// flags - Flags that help describe the property.

void DebuggerScope::AddProperty(RegSlot location, Js::PropertyId propertyId, DebuggerScopePropertyFlags flags)

{

DebuggerScopeProperty scopeProperty;

scopeProperty.location = location;

scopeProperty.propId = propertyId;

// This offset is uninitialized until the property is initialized (with a ld opcode, for example).

scopeProperty.byteCodeInitializationOffset = Constants::InvalidByteCodeOffset;

scopeProperty.flags = flags;

// Delay allocate the property list so we don't take up memory if there are no properties in this scope.

// Scopes are created during non-debug mode as well so we want to keep them as small as possible.

this->EnsurePropertyListIsAllocated();

// The property doesn't exist yet, so add it.

this->scopeProperties->Add(scopeProperty);

}

bool DebuggerScope::GetPropertyIndex(Js::PropertyId propertyId, int& index)

{

if (!this->HasProperties())

{

index = -1;

return false;

}

bool found = this->scopeProperties->MapUntil( [&](int i, const DebuggerScopeProperty& scopeProperty) {

if(scopeProperty.propId == propertyId)

{

index = scopeProperty.location;

return true;

}

return false;

});

if(!found)

{

return false;

}

return true;

}

#if DBG

void DebuggerScope::Dump()

{

int indent = (GetScopeDepth() - 1) \* 4;

Output::Print(indent, L"Begin scope: Address: %p Type: %s Location: %d Sibling: %p Range: [%d, %d]\n ", this, GetDebuggerScopeTypeString(scopeType), scopeLocation, this->siblingScope, range.begin, range.end);

if (this->HasProperties())

{

this->scopeProperties->Map( [=] (int i, Js::DebuggerScopeProperty& scopeProperty) {

Output::Print(indent, L"%s(%d) Location: %d Const: %s Initialized: %d\n", ThreadContext::GetContextForCurrentThread()->GetPropertyName(scopeProperty.propId)->GetBuffer(),

scopeProperty.propId, scopeProperty.location, scopeProperty.IsConst() ? L"true": L"false", scopeProperty.byteCodeInitializationOffset);

});

}

Output::Print(L"\n");

}

// Returns the debugger scope type in string format.

PCWSTR DebuggerScope::GetDebuggerScopeTypeString(DiagExtraScopesType scopeType)

{

switch (scopeType)

{

case DiagExtraScopesType::DiagBlockScopeDirect:

return L"DiagBlockScopeDirect";

case DiagExtraScopesType::DiagBlockScopeInObject:

return L"DiagBlockScopeInObject";

case DiagExtraScopesType::DiagBlockScopeInSlot:

return L"DiagBlockScopeInSlot";

case DiagExtraScopesType::DiagBlockScopeRangeEnd:

return L"DiagBlockScopeRangeEnd";

case DiagExtraScopesType::DiagCatchScopeDirect:

return L"DiagCatchScopeDirect";

case DiagExtraScopesType::DiagCatchScopeInObject:

return L"DiagCatchScopeInObject";

case DiagExtraScopesType::DiagCatchScopeInSlot:

return L"DiagCatchScopeInSlot";

case DiagExtraScopesType::DiagUnknownScope:

return L"DiagUnknownScope";

case DiagExtraScopesType::DiagWithScope:

return L"DiagWithScope";

default:

AssertMsg(false, "Missing a debug scope type.");

return L"";

}

}

#endif

// Updates the current offset of where the property is first initialized. This is used to

// detect whether or not a property is in a dead zone when broken in the debugger.

// location - The slot array index or register slot location of where the property is stored.

// propertyId - The property ID of the property.

// byteCodeOffset - The offset to set the initialization point at.

// isFunctionDeclaration - Whether or not the property is a function declaration or not. Used for verification.

// <returns> - True if the property was found and updated for the current scope, else false.

bool DebuggerScope::UpdatePropertyInitializationOffset(

RegSlot location,

Js::PropertyId propertyId,

int byteCodeOffset,

bool isFunctionDeclaration /\*= false\*/)

{

if (UpdatePropertyInitializationOffsetInternal(location, propertyId, byteCodeOffset, isFunctionDeclaration))

{

return true;

}

if (siblingScope != nullptr && siblingScope->UpdatePropertyInitializationOffsetInternal(location, propertyId, byteCodeOffset, isFunctionDeclaration))

{

return true;

}

return false;

}

bool DebuggerScope::UpdatePropertyInitializationOffsetInternal(

RegSlot location,

Js::PropertyId propertyId,

int byteCodeOffset,

bool isFunctionDeclaration /\*= false\*/)

{

if (scopeProperties == nullptr)

{

return false;

}

for (int i = 0; i < scopeProperties->Count(); ++i)

{

DebuggerScopeProperty propertyItem = scopeProperties->Item(i);

if (propertyItem.propId == propertyId && propertyItem.location == location)

{

if (propertyItem.byteCodeInitializationOffset == Constants::InvalidByteCodeOffset)

{

propertyItem.byteCodeInitializationOffset = byteCodeOffset;

scopeProperties->SetExistingItem(i, propertyItem);

}

#if DBG

else

{

// If the bytecode initialization offset is not Constants::InvalidByteCodeOffset,

// it means we have two or more functions declared in the same scope with the same name

// and one has already been marked. We track each location with a property entry

// on the debugging side (when calling DebuggerScope::AddProperty()) as opposed to scanning

// and checking if the property already exists each time we add in order to avoid duplicates.

AssertMsg(isFunctionDeclaration, "Only function declarations can be defined more than once in the same scope with the same name.");

AssertMsg(propertyItem.byteCodeInitializationOffset == byteCodeOffset, "The bytecode offset for all function declarations should be identical for this scope.");

}

#endif // DBG

return true;

}

}

return false;

}

// Updates the debugger scopes fields due to a regeneration of bytecode (happens during debugger attach or detach, for

// example).

void DebuggerScope::UpdateDueToByteCodeRegeneration(DiagExtraScopesType scopeType, int start, RegSlot scopeLocation)

{

#if DBG

if (this->scopeType != Js::DiagUnknownScope)

{

// If the scope is unknown, it was deserialized without a scope type. Otherwise, it should not have changed.

// The scope type can change on a re-parse in certain scenarios related to eval detection in legacy mode -> Winblue: 272122

AssertMsg(this->scopeType == scopeType, "The debugger scope type should not have changed when generating bytecode again.");

}

#endif // DBG

this->scopeType = scopeType;

this->SetBegin(start);

if(this->scopeProperties)

{

this->scopeProperties->Clear();

}

// Reset the scope location as it may have changed during bytecode generation from the last run.

this->SetScopeLocation(scopeLocation);

if (siblingScope)

{

// If we had a sibling scope during initial parsing, clear it now so that it will be reset

// when it is retrieved during this bytecode generation pass, in GetSiblingScope().

// GetSiblingScope() will ensure that the FunctionBody currentDebuggerScopeIndex value is

// updated accordingly to account for future scopes coming after the sibling.

// Calling of GetSiblingScope() will happen when register properties are added to this scope

// via TrackRegisterPropertyForDebugger().

siblingScope = nullptr;

}

}

void DebuggerScope::UpdatePropertiesInForInOrOfCollectionScope()

{

if (this->scopeProperties != nullptr)

{

this->scopeProperties->All([&](Js::DebuggerScopeProperty& propertyItem)

{

propertyItem.flags |= DebuggerScopePropertyFlags\_ForInOrOfCollection;

return true;

});

}

}

void DebuggerScope::EnsurePropertyListIsAllocated()

{

if (this->scopeProperties == nullptr)

{

this->scopeProperties = RecyclerNew(this->recycler, DebuggerScopePropertyList, this->recycler);

}

}

// Checks if the passed in ByteCodeGenerator offset is in this scope's being/end range.

bool DebuggerScope::IsOffsetInScope(int offset) const

{

Assert(this->range.end != -1);

return this->range.Includes(offset);

}

// Determines if the DebuggerScope contains a property with the passed in ID and

// location in the internal property list.

// propertyId - The ID of the property to search for.

// location - The slot array index or register to search for.

// outScopeProperty - Optional parameter that will return the property, if found.

bool DebuggerScope::Contains(Js::PropertyId propertyId, RegSlot location) const

{

DebuggerScopeProperty tempProperty;

return TryGetProperty(propertyId, location, &tempProperty);

}

// Gets whether or not the scope is a block scope (non-catch or with).

bool DebuggerScope::IsBlockScope() const

{

AssertMsg(this->scopeType != Js::DiagBlockScopeRangeEnd, "Debugger scope type should never be set to range end - only reserved for marking the end of a scope (not persisted).");

return this->scopeType == Js::DiagBlockScopeDirect

|| this->scopeType == Js::DiagBlockScopeInObject

|| this->scopeType == Js::DiagBlockScopeInSlot

|| this->scopeType == Js::DiagBlockScopeRangeEnd;

}

// Gets whether or not the scope is a catch block scope.

bool DebuggerScope::IsCatchScope() const

{

return this->scopeType == Js::DiagCatchScopeDirect

|| this->scopeType == Js::DiagCatchScopeInObject

|| this->scopeType == Js::DiagCatchScopeInSlot;

}

// Gets whether or not the scope is a with block scope.

bool DebuggerScope::IsWithScope() const

{

return this->scopeType == Js::DiagWithScope;

}

// Gets whether or not the scope is a slot array scope.

bool DebuggerScope::IsSlotScope() const

{

return this->scopeType == Js::DiagBlockScopeInSlot

|| this->scopeType == Js::DiagCatchScopeInSlot;

}

// Gets whether or not the scope has any properties in it.

bool DebuggerScope::HasProperties() const

{

return this->scopeProperties && this->scopeProperties->Count() > 0;

}

// Checks if this scope is an ancestor of the passed in scope.

bool DebuggerScope::IsAncestorOf(const DebuggerScope\* potentialChildScope)

{

if (potentialChildScope == nullptr)

{

// If the child scope is null, it represents the global scope which

// cannot be a child of anything.

return false;

}

const DebuggerScope\* currentScope = potentialChildScope;

while (currentScope)

{

if (currentScope->GetParentScope() == this)

{

return true;

}

currentScope = currentScope->GetParentScope();

}

return false;

}

// Checks if all properties of the scope are currently in a dead zone given the specified offset.

bool DebuggerScope::AreAllPropertiesInDeadZone(int byteCodeOffset) const

{

if (!this->HasProperties())

{

return false;

}

return this->scopeProperties->All([&](Js::DebuggerScopeProperty& propertyItem)

{

return propertyItem.IsInDeadZone(byteCodeOffset);

});

}

// Attempts to get the specified property. Returns true if the property was copied to the structure; false otherwise.

bool DebuggerScope::TryGetProperty(Js::PropertyId propertyId, RegSlot location, DebuggerScopeProperty\* outScopeProperty) const

{

Assert(outScopeProperty);

if (scopeProperties == nullptr)

{

return false;

}

for (int i = 0; i < scopeProperties->Count(); ++i)

{

DebuggerScopeProperty propertyItem = scopeProperties->Item(i);

if (propertyItem.propId == propertyId && propertyItem.location == location)

{

\*outScopeProperty = propertyItem;

return true;

}

}

return false;

}

bool DebuggerScope::TryGetValidProperty(Js::PropertyId propertyId, RegSlot location, int offset, DebuggerScopeProperty\* outScopeProperty, bool\* isInDeadZone) const

{

if (TryGetProperty(propertyId, location, outScopeProperty))

{

if (IsOffsetInScope(offset))

{

if (isInDeadZone != nullptr)

{

\*isInDeadZone = outScopeProperty->IsInDeadZone(offset);

}

return true;

}

}

return false;

}

void DebuggerScope::SetBegin(int begin)

{

range.begin = begin;

if (siblingScope != nullptr)

{

siblingScope->SetBegin(begin);

}

}

void DebuggerScope::SetEnd(int end)

{

range.end = end;

if (siblingScope != nullptr)

{

siblingScope->SetEnd(end);

}

}

// Finds the common ancestor scope between this scope and the passed in scope.

// Returns nullptr if the scopes are part of different trees.

DebuggerScope\* DebuggerScope::FindCommonAncestor(DebuggerScope\* debuggerScope)

{

AnalysisAssert(debuggerScope);

if (this == debuggerScope)

{

return debuggerScope;

}

if (this->IsAncestorOf(debuggerScope))

{

return this;

}

if (debuggerScope->IsAncestorOf(this))

{

return debuggerScope;

}

DebuggerScope\* firstNode = this;

DebuggerScope\* secondNode = debuggerScope;

int firstDepth = firstNode->GetScopeDepth();

int secondDepth = secondNode->GetScopeDepth();

// Calculate the depth difference in order to bring the deep node up to the sibling

// level of the shorter node.

int depthDifference = abs(firstDepth - secondDepth);

DebuggerScope\*& nodeToBringUp = firstDepth > secondDepth ? firstNode : secondNode;

while (depthDifference > 0)

{

AnalysisAssert(nodeToBringUp);

nodeToBringUp = nodeToBringUp->GetParentScope();

--depthDifference;

}

// Move up the tree and see where the nodes meet.

while (firstNode && secondNode)

{

if (firstNode == secondNode)

{

return firstNode;

}

firstNode = firstNode->GetParentScope();

secondNode = secondNode->GetParentScope();

}

// The nodes are not part of the same scope tree.

return nullptr;

}

// Gets the depth of the scope in the parent link tree.

int DebuggerScope::GetScopeDepth() const

{

int depth = 0;

const DebuggerScope\* currentDebuggerScope = this;

while (currentDebuggerScope)

{

currentDebuggerScope = currentDebuggerScope->GetParentScope();

++depth;

}

return depth;

}

bool ScopeObjectChain::TryGetDebuggerScopePropertyInfo(PropertyId propertyId, RegSlot location, int offset, bool\* isPropertyInDebuggerScope, bool \*isConst, bool\* isInDeadZone)

{

Assert(pScopeChain);

Assert(isPropertyInDebuggerScope);

Assert(isConst);

\*isPropertyInDebuggerScope = false;

\*isConst = false;

// Search through each block scope until we find the current scope. If the register was found

// in any of the scopes going down until we reach the scope of the debug break, then it's in scope.

// if found but not in the scope, the out param will be updated (since it is actually a let or const), so that caller can make a call accordingly.

for (int i = 0; i < pScopeChain->Count(); i++)

{

Js::DebuggerScope \*debuggerScope = pScopeChain->Item(i);

DebuggerScopeProperty debuggerScopeProperty;

if (debuggerScope->TryGetProperty(propertyId, location, &debuggerScopeProperty))

{

bool isOffsetInScope = debuggerScope->IsOffsetInScope(offset);

// For the Object scope, all the properties will have the same location (-1) so they can match. Use further check below to determine the propertyInDebuggerScope

\*isPropertyInDebuggerScope = isOffsetInScope || !debuggerScope->IsBlockObjectScope();

if (isOffsetInScope)

{

if (isInDeadZone != nullptr)

{

\*isInDeadZone = debuggerScopeProperty.IsInDeadZone(offset);

}

\*isConst = debuggerScopeProperty.IsConst();

return true;

}

}

}

return false;

}

void FunctionBody::AllocateInlineCache()

{

Assert(this->inlineCaches == nullptr);

uint isInstInlineCacheStart = this->GetInlineCacheCount();

uint totalCacheCount = isInstInlineCacheStart + isInstInlineCacheCount;

if (totalCacheCount != 0)

{

// Root object inline cache are not leaf

void \*\* inlineCaches = RecyclerNewArrayZ(this->m\_scriptContext->GetRecycler(),

void\*, totalCacheCount);

#if DBG

this->m\_inlineCacheTypes = RecyclerNewArrayLeafZ(this->m\_scriptContext->GetRecycler(),

byte, totalCacheCount);

#endif

uint i = 0;

uint plainInlineCacheEnd = rootObjectLoadInlineCacheStart;

\_\_analysis\_assume(plainInlineCacheEnd <= totalCacheCount);

for (; i < plainInlineCacheEnd; i++)

{

inlineCaches[i] = AllocatorNewZ(InlineCacheAllocator,

this->m\_scriptContext->GetInlineCacheAllocator(), InlineCache);

}

Js::RootObjectBase \* rootObject = this->GetRootObject();

ThreadContext \* threadContext = this->GetScriptContext()->GetThreadContext();

uint rootObjectLoadInlineCacheEnd = rootObjectLoadMethodInlineCacheStart;

\_\_analysis\_assume(rootObjectLoadInlineCacheEnd <= totalCacheCount);

for (; i < rootObjectLoadInlineCacheEnd; i++)

{

inlineCaches[i] = rootObject->GetInlineCache(

threadContext->GetPropertyName(this->GetPropertyIdFromCacheId(i)), false, false);

}

uint rootObjectLoadMethodInlineCacheEnd = rootObjectStoreInlineCacheStart;

\_\_analysis\_assume(rootObjectLoadMethodInlineCacheEnd <= totalCacheCount);

for (; i < rootObjectLoadMethodInlineCacheEnd; i++)

{

inlineCaches[i] = rootObject->GetInlineCache(

threadContext->GetPropertyName(this->GetPropertyIdFromCacheId(i)), true, false);

}

uint rootObjectStoreInlineCacheEnd = isInstInlineCacheStart;

\_\_analysis\_assume(rootObjectStoreInlineCacheEnd <= totalCacheCount);

for (; i < rootObjectStoreInlineCacheEnd; i++)

{

#pragma prefast(suppress:6386, "The analysis assume didn't help prefast figure out this is in range")

inlineCaches[i] = rootObject->GetInlineCache(

threadContext->GetPropertyName(this->GetPropertyIdFromCacheId(i)), false, true);

}

for (; i < totalCacheCount; i++)

{

inlineCaches[i] = AllocatorNewStructZ(IsInstInlineCacheAllocator,

this->m\_scriptContext->GetIsInstInlineCacheAllocator(), IsInstInlineCache);

}

#if DBG

this->m\_inlineCacheTypes = RecyclerNewArrayLeafZ(this->m\_scriptContext->GetRecycler(),

byte, totalCacheCount);

#endif

this->inlineCaches = inlineCaches;

}

}

InlineCache \*FunctionBody::GetInlineCache(uint index)

{

Assert(this->inlineCaches != nullptr);

Assert(index < this->GetInlineCacheCount());

#if DBG

Assert(this->m\_inlineCacheTypes[index] == InlineCacheTypeNone ||

this->m\_inlineCacheTypes[index] == InlineCacheTypeInlineCache);

this->m\_inlineCacheTypes[index] = InlineCacheTypeInlineCache;

#endif

return reinterpret\_cast<InlineCache \*>(this->inlineCaches[index]);

}

bool FunctionBody::CanFunctionObjectHaveInlineCaches()

{

if (this->DoStackNestedFunc() || this->IsGenerator())

{

return false;

}

uint totalCacheCount = this->GetInlineCacheCount() + this->GetIsInstInlineCacheCount();

if (PHASE\_FORCE(Js::ScriptFunctionWithInlineCachePhase, this) && totalCacheCount > 0)

{

return true;

}

// Only have inline caches on function object for possible inlining candidates.

// Since we don't know the size of the top function, check against the maximum possible inline threshold

// Negative inline byte code size threshold will disable inline cache on function object.

const int byteCodeSizeThreshold = CONFIG\_FLAG(InlineThreshold) + CONFIG\_FLAG(InlineThresholdAdjustCountInSmallFunction);

if (byteCodeSizeThreshold < 0 || this->GetByteCodeWithoutLDACount() > (uint)byteCodeSizeThreshold)

{

return false;

}

// Negative FuncObjectInlineCacheThreshold will disable inline cache on function object.

if (CONFIG\_FLAG(FuncObjectInlineCacheThreshold) < 0 || totalCacheCount > (uint)CONFIG\_FLAG(FuncObjectInlineCacheThreshold) || totalCacheCount == 0)

{

return false;

}

return true;

}

void\*\* FunctionBody::GetInlineCaches()

{

return this->inlineCaches;

}

#if DBG

byte\* FunctionBody::GetInlineCacheTypes()

{

return this->m\_inlineCacheTypes;

}

#endif

IsInstInlineCache \*FunctionBody::GetIsInstInlineCache(uint index)

{

Assert(this->inlineCaches != nullptr);

Assert(index < GetIsInstInlineCacheCount());

index += this->GetInlineCacheCount();

#if DBG

Assert(this->m\_inlineCacheTypes[index] == InlineCacheTypeNone ||

this->m\_inlineCacheTypes[index] == InlineCacheTypeIsInst);

this->m\_inlineCacheTypes[index] = InlineCacheTypeIsInst;

#endif

return reinterpret\_cast<IsInstInlineCache \*>(this->inlineCaches[index]);

}

PolymorphicInlineCache \* FunctionBody::GetPolymorphicInlineCache(uint index)

{

return this->polymorphicInlineCaches.GetInlineCache(this, index);

}

PolymorphicInlineCache \* FunctionBody::CreateNewPolymorphicInlineCache(uint index, PropertyId propertyId, InlineCache \* inlineCache)

{

Assert(GetPolymorphicInlineCache(index) == nullptr);

// Only create polymorphic inline caches for non-root inline cache indexes

if (index < rootObjectLoadInlineCacheStart

#if DBG

&& !PHASE\_OFF1(Js::PolymorphicInlineCachePhase)

#endif

)

{

PolymorphicInlineCache \* polymorphicInlineCache = CreatePolymorphicInlineCache(index, PolymorphicInlineCache::GetInitialSize());

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (PHASE\_VERBOSE\_TRACE1(Js::PolymorphicInlineCachePhase))

{

this->DumpFullFunctionName();

Output::Print(L": New PIC, index = %d, size = %d\n", index, PolymorphicInlineCache::GetInitialSize());

}

#endif

#if PHASE\_PRINT\_INTRUSIVE\_TESTTRACE1

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

#endif

PHASE\_PRINT\_INTRUSIVE\_TESTTRACE1(

Js::PolymorphicInlineCachePhase,

L"TestTrace PIC: New, Function %s (%s), 0x%x, index = %d, size = %d\n", this->GetDisplayName(), this->GetDebugNumberSet(debugStringBuffer), polymorphicInlineCache, index, PolymorphicInlineCache::GetInitialSize());

uint indexInPolymorphicCache = polymorphicInlineCache->GetInlineCacheIndexForType(inlineCache->GetType());

inlineCache->CopyTo(propertyId, m\_scriptContext, &(polymorphicInlineCache->GetInlineCaches()[indexInPolymorphicCache]));

polymorphicInlineCache->UpdateInlineCachesFillInfo(indexInPolymorphicCache, true /\*set\*/);

return polymorphicInlineCache;

}

return nullptr;

}

PolymorphicInlineCache \* FunctionBody::CreateBiggerPolymorphicInlineCache(uint index, PropertyId propertyId)

{

PolymorphicInlineCache \* polymorphicInlineCache = GetPolymorphicInlineCache(index);

Assert(polymorphicInlineCache && polymorphicInlineCache->CanAllocateBigger());

uint16 polymorphicInlineCacheSize = polymorphicInlineCache->GetSize();

uint16 newPolymorphicInlineCacheSize = PolymorphicInlineCache::GetNextSize(polymorphicInlineCacheSize);

Assert(newPolymorphicInlineCacheSize > polymorphicInlineCacheSize);

PolymorphicInlineCache \* newPolymorphicInlineCache = CreatePolymorphicInlineCache(index, newPolymorphicInlineCacheSize);

polymorphicInlineCache->CopyTo(propertyId, m\_scriptContext, newPolymorphicInlineCache);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (PHASE\_VERBOSE\_TRACE1(Js::PolymorphicInlineCachePhase))

{

this->DumpFullFunctionName();

Output::Print(L": Bigger PIC, index = %d, oldSize = %d, newSize = %d\n", index, polymorphicInlineCacheSize, newPolymorphicInlineCacheSize);

}

#endif

#if PHASE\_PRINT\_INTRUSIVE\_TESTTRACE1

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

#endif

PHASE\_PRINT\_INTRUSIVE\_TESTTRACE1(

Js::PolymorphicInlineCachePhase,

L"TestTrace PIC: Bigger, Function %s (%s), 0x%x, index = %d, size = %d\n", this->GetDisplayName(), this->GetDebugNumberSet(debugStringBuffer), newPolymorphicInlineCache, index, newPolymorphicInlineCacheSize);

return newPolymorphicInlineCache;

}

void FunctionBody::ResetInlineCaches()

{

isInstInlineCacheCount = inlineCacheCount = rootObjectLoadInlineCacheStart = rootObjectStoreInlineCacheStart = 0;

this->inlineCaches = nullptr;

this->polymorphicInlineCaches.Reset();

}

PolymorphicInlineCache \* FunctionBody::CreatePolymorphicInlineCache(uint index, uint16 size)

{

Recycler \* recycler = this->m\_scriptContext->GetRecycler();

PolymorphicInlineCache \* newPolymorphicInlineCache = PolymorphicInlineCache::New(size, this);

this->polymorphicInlineCaches.SetInlineCache(recycler, this, index, newPolymorphicInlineCache);

return newPolymorphicInlineCache;

}

uint FunctionBody::NewObjectLiteral()

{

Assert(objLiteralTypes == nullptr);

return objLiteralCount++;

}

DynamicType \*\* FunctionBody::GetObjectLiteralTypeRef(uint index)

{

Assert(index < objLiteralCount);

Assert(objLiteralTypes != nullptr);

return objLiteralTypes + index;

}

void FunctionBody::AllocateObjectLiteralTypeArray()

{

Assert(objLiteralTypes == nullptr);

if (objLiteralCount == 0)

{

return;

}

objLiteralTypes = RecyclerNewArrayZ(this->GetScriptContext()->GetRecycler(), DynamicType \*, objLiteralCount);

}

uint FunctionBody::NewLiteralRegex()

{

Assert(!this->literalRegexes);

return literalRegexCount++;

}

uint FunctionBody::GetLiteralRegexCount() const

{

return literalRegexCount;

}

void FunctionBody::AllocateLiteralRegexArray()

{

Assert(!this->literalRegexes);

if (literalRegexCount == 0)

{

return;

}

this->literalRegexes =

RecyclerNewArrayZ(m\_scriptContext->GetRecycler(), UnifiedRegex::RegexPattern \*, literalRegexCount);

}

#ifndef TEMP\_DISABLE\_ASMJS

AsmJsFunctionInfo\* FunctionBody::AllocateAsmJsFunctionInfo()

{

Assert( !this->asmJsFunctionInfo );

this->asmJsFunctionInfo = RecyclerNew( m\_scriptContext->GetRecycler(), AsmJsFunctionInfo );

return this->asmJsFunctionInfo;

}

AsmJsModuleInfo\* FunctionBody::AllocateAsmJsModuleInfo()

{

Assert( !this->asmJsModuleInfo );

Recycler\* rec = m\_scriptContext->GetRecycler();

this->asmJsModuleInfo = RecyclerNew( rec, AsmJsModuleInfo, rec );

return this->asmJsModuleInfo;

}

#endif

UnifiedRegex::RegexPattern \*FunctionBody::GetLiteralRegex(const uint index)

{

Assert(index < literalRegexCount);

Assert(this->literalRegexes);

return this->literalRegexes[index];

}

void FunctionBody::SetLiteralRegex(const uint index, UnifiedRegex::RegexPattern \*const pattern)

{

Assert(index < literalRegexCount);

Assert(this->literalRegexes);

if (this->literalRegexes[index] && this->literalRegexes[index] == pattern)

{

return;

}

Assert(!this->literalRegexes[index]);

this->literalRegexes[index] = pattern;

}

void FunctionBody::ResetObjectLiteralTypes()

{

this->objLiteralTypes = nullptr;

this->objLiteralCount = 0;

}

void FunctionBody::ResetLiteralRegexes()

{

literalRegexCount = 0;

this->literalRegexes = nullptr;

}

void FunctionBody::ResetProfileIds()

{

#if ENABLE\_PROFILE\_INFO

Assert(!HasDynamicProfileInfo()); // profile data relies on the profile ID counts; it should not have been created yet

Assert(!this->m\_codeGenRuntimeData); // relies on 'profiledCallSiteCount'

profiledCallSiteCount = 0;

profiledArrayCallSiteCount = 0;

profiledReturnTypeCount = 0;

profiledSlotCount = 0;

profiledLdElemCount = 0;

profiledStElemCount = 0;

#endif

}

void FunctionBody::ResetByteCodeGenState()

{

// Byte code generation failed for this function. Revert any intermediate state being tracked in the function body, in

// case byte code generation is attempted again for this function body.

ResetInlineCaches();

ResetObjectLiteralTypes();

ResetLiteralRegexes();

ResetLoops();

ResetProfileIds();

m\_firstTmpReg = Constants::NoRegister;

localClosureRegister = Constants::NoRegister;

localFrameDisplayRegister = Constants::NoRegister;

envRegister = Constants::NoRegister;

thisRegisterForEventHandler = Constants::NoRegister;

firstInnerScopeRegister = Constants::NoRegister;

funcExprScopeRegister = Constants::NoRegister;

innerScopeCount = 0;

hasCachedScopePropIds = false;

m\_constCount = 0;

this->m\_constTable = nullptr;

this->byteCodeBlock = nullptr;

// There is other state that is set by the byte code generator but the state should be the same each time byte code

// generation is done for the function, so it doesn't need to be reverted

}

void FunctionBody::ResetByteCodeGenVisitState()

{

// This function body is about to be visited by the byte code generator after defer-parsing it. Since the previous visit

// pass may have failed, we need to restore state that is tracked on the function body by the visit pass.

ResetLiteralRegexes();

}

#if ENABLE\_NATIVE\_CODEGEN

const FunctionCodeGenRuntimeData \*FunctionBody::GetInlineeCodeGenRuntimeData(const ProfileId profiledCallSiteId) const

{

Assert(profiledCallSiteId < profiledCallSiteCount);

return this->m\_codeGenRuntimeData ? this->m\_codeGenRuntimeData[profiledCallSiteId] : nullptr;

}

const FunctionCodeGenRuntimeData \*FunctionBody::GetInlineeCodeGenRuntimeDataForTargetInlinee(const ProfileId profiledCallSiteId, Js::FunctionBody \*inlineeFuncBody) const

{

Assert(profiledCallSiteId < profiledCallSiteCount);

if (!this->m\_codeGenRuntimeData)

{

return nullptr;

}

const FunctionCodeGenRuntimeData \*runtimeData = this->m\_codeGenRuntimeData[profiledCallSiteId];

while (runtimeData && runtimeData->GetFunctionBody() != inlineeFuncBody)

{

runtimeData = runtimeData->GetNext();

}

return runtimeData;

}

FunctionCodeGenRuntimeData \*FunctionBody::EnsureInlineeCodeGenRuntimeData(

Recycler \*const recycler,

\_\_in\_range(0, profiledCallSiteCount - 1) const ProfileId profiledCallSiteId,

FunctionBody \*const inlinee)

{

Assert(recycler);

Assert(profiledCallSiteId < profiledCallSiteCount);

Assert(inlinee);

if(!this->m\_codeGenRuntimeData)

{

const auto codeGenRuntimeData = RecyclerNewArrayZ(recycler, FunctionCodeGenRuntimeData \*, profiledCallSiteCount);

this->m\_codeGenRuntimeData = codeGenRuntimeData;

}

const auto inlineeData = this->m\_codeGenRuntimeData[profiledCallSiteId];

if(!inlineeData)

{

return this->m\_codeGenRuntimeData[profiledCallSiteId] = RecyclerNew(recycler, FunctionCodeGenRuntimeData, inlinee);

}

// Find the right code gen runtime data

FunctionCodeGenRuntimeData \*next = inlineeData;

while(next && (next->GetFunctionBody() != inlinee))

{

next = next->GetNext();

}

if (next)

{

return next;

}

FunctionCodeGenRuntimeData \*runtimeData = RecyclerNew(recycler, FunctionCodeGenRuntimeData, inlinee);

runtimeData->SetupRuntimeDataChain(inlineeData);

return this->m\_codeGenRuntimeData[profiledCallSiteId] = runtimeData;

}

const FunctionCodeGenRuntimeData \*FunctionBody::GetLdFldInlineeCodeGenRuntimeData(const uint inlineCacheIndex) const

{

Assert(inlineCacheIndex < inlineCacheCount);

return this->m\_codeGenGetSetRuntimeData ? this->m\_codeGenGetSetRuntimeData[inlineCacheIndex] : nullptr;

}

FunctionCodeGenRuntimeData \*FunctionBody::EnsureLdFldInlineeCodeGenRuntimeData(

Recycler \*const recycler,

\_\_in\_range(0, this->inlineCacheCount - 1) const uint inlineCacheIndex,

FunctionBody \*const inlinee)

{

Assert(recycler);

Assert(inlineCacheIndex < this->GetInlineCacheCount());

Assert(inlinee);

if(!this->m\_codeGenGetSetRuntimeData)

{

const auto codeGenRuntimeData = RecyclerNewArrayZ(recycler, FunctionCodeGenRuntimeData \*, this->GetInlineCacheCount());

this->m\_codeGenGetSetRuntimeData = codeGenRuntimeData;

}

const auto inlineeData = this->m\_codeGenGetSetRuntimeData[inlineCacheIndex];

if(inlineeData)

{

return inlineeData;

}

return this->m\_codeGenGetSetRuntimeData[inlineCacheIndex] = RecyclerNew(recycler, FunctionCodeGenRuntimeData, inlinee);

}

#endif

void FunctionBody::AllocateLoopHeaders()

{

Assert(this->loopHeaderArray == nullptr);

if (loopCount != 0)

{

this->loopHeaderArray = RecyclerNewArrayZ(this->m\_scriptContext->GetRecycler(), LoopHeader, loopCount);

for (uint i = 0; i < loopCount; i++)

{

this->loopHeaderArray[i].Init(this);

}

}

}

void FunctionBody::ReleaseLoopHeaders()

{

#if ENABLE\_NATIVE\_CODEGEN

this->MapLoopHeaders([](uint loopNumber, LoopHeader \* loopHeader)

{

loopHeader->ReleaseEntryPoints();

});

#endif

}

void FunctionBody::ResetLoops()

{

loopCount = 0;

this->loopHeaderArray = nullptr;

}

void FunctionBody::RestoreOldDefaultEntryPoint(FunctionEntryPointInfo\* oldEntryPointInfo,

JavascriptMethod oldOriginalEntryPoint,

FunctionEntryPointInfo\* newEntryPointInfo)

{

Assert(newEntryPointInfo);

this->SetDefaultFunctionEntryPointInfo(oldEntryPointInfo, oldOriginalEntryPoint);

this->entryPoints->RemoveAt(newEntryPointInfo->entryPointIndex);

}

FunctionEntryPointInfo\* FunctionBody::CreateNewDefaultEntryPoint()

{

Recycler \*const recycler = this->m\_scriptContext->GetRecycler();

const JavascriptMethod currentThunk = m\_scriptContext->CurrentThunk;

void\* validationCookie = nullptr;

#if ENABLE\_NATIVE\_CODEGEN

validationCookie = (void\*)m\_scriptContext->GetNativeCodeGenerator();

#endif

FunctionEntryPointInfo \*const entryPointInfo =

RecyclerNewFinalized(

recycler,

FunctionEntryPointInfo,

this,

currentThunk,

m\_scriptContext->GetThreadContext(),

validationCookie);

AddEntryPointToEntryPointList(entryPointInfo);

{

// Allocations in this region may trigger expiry and cause unexpected changes to state

AUTO\_NO\_EXCEPTION\_REGION;

FunctionEntryPointInfo \*const simpleJitEntryPointInfo = GetSimpleJitEntryPointInfo();

Js::JavascriptMethod originalEntryPoint, directEntryPoint;

if(simpleJitEntryPointInfo && GetExecutionMode() == ExecutionMode::FullJit)

{

directEntryPoint =

originalEntryPoint = reinterpret\_cast<Js::JavascriptMethod>(simpleJitEntryPointInfo->GetNativeAddress());

}

else

{

#if DYNAMIC\_INTERPRETER\_THUNK

// If the dynamic interpreter thunk hasn't been created yet, then the entry point can be set to

// the default entry point. Otherwise, since the new default entry point is being created to

// move back to the interpreter, the original entry point is going to be the dynamic interpreter thunk

originalEntryPoint =

m\_dynamicInterpreterThunk

? static\_cast<JavascriptMethod>(InterpreterThunkEmitter::ConvertToEntryPoint(m\_dynamicInterpreterThunk))

: DefaultEntryThunk;

#else

originalEntryPoint = DefaultEntryThunk;

#endif

directEntryPoint = currentThunk == DefaultEntryThunk ? originalEntryPoint : currentThunk;

}

entryPointInfo->address = directEntryPoint;

SetDefaultFunctionEntryPointInfo(entryPointInfo, originalEntryPoint);

}

return entryPointInfo;

}

LoopHeader \*FunctionBody::GetLoopHeader(uint index) const

{

Assert(this->loopHeaderArray != nullptr);

Assert(index < loopCount);

return &this->loopHeaderArray[index];

}

FunctionEntryPointInfo \*FunctionBody::GetSimpleJitEntryPointInfo() const

{

return simpleJitEntryPointInfo;

}

void FunctionBody::SetSimpleJitEntryPointInfo(FunctionEntryPointInfo \*const entryPointInfo)

{

simpleJitEntryPointInfo = entryPointInfo;

}

void FunctionBody::VerifyExecutionMode(const ExecutionMode executionMode) const

{

#if DBG

Assert(initializedExecutionModeAndLimits);

Assert(executionMode < ExecutionMode::Count);

switch(executionMode)

{

case ExecutionMode::Interpreter:

Assert(!DoInterpreterProfile());

break;

case ExecutionMode::AutoProfilingInterpreter:

Assert(DoInterpreterProfile());

Assert(DoInterpreterAutoProfile());

break;

case ExecutionMode::ProfilingInterpreter:

Assert(DoInterpreterProfile());

break;

case ExecutionMode::SimpleJit:

Assert(DoSimpleJit());

break;

case ExecutionMode::FullJit:

Assert(DoFullJit());

break;

default:

Assert(false);

\_\_assume(false);

}

#endif

}

ExecutionMode FunctionBody::GetDefaultInterpreterExecutionMode() const

{

if(!DoInterpreterProfile())

{

VerifyExecutionMode(ExecutionMode::Interpreter);

return ExecutionMode::Interpreter;

}

if(DoInterpreterAutoProfile())

{

VerifyExecutionMode(ExecutionMode::AutoProfilingInterpreter);

return ExecutionMode::AutoProfilingInterpreter;

}

VerifyExecutionMode(ExecutionMode::ProfilingInterpreter);

return ExecutionMode::ProfilingInterpreter;

}

ExecutionMode FunctionBody::GetExecutionMode() const

{

VerifyExecutionMode(executionMode);

return executionMode;

}

ExecutionMode FunctionBody::GetInterpreterExecutionMode(const bool isPostBailout)

{

Assert(initializedExecutionModeAndLimits);

if(isPostBailout && DoInterpreterProfile())

{

return ExecutionMode::ProfilingInterpreter;

}

switch(GetExecutionMode())

{

case ExecutionMode::Interpreter:

case ExecutionMode::AutoProfilingInterpreter:

case ExecutionMode::ProfilingInterpreter:

return GetExecutionMode();

case ExecutionMode::SimpleJit:

if(IsNewSimpleJit())

{

return GetDefaultInterpreterExecutionMode();

}

// fall through

case ExecutionMode::FullJit:

{

const ExecutionMode executionMode =

DoInterpreterProfile() ? ExecutionMode::ProfilingInterpreter : ExecutionMode::Interpreter;

VerifyExecutionMode(executionMode);

return executionMode;

}

default:

Assert(false);

\_\_assume(false);

}

}

void FunctionBody::SetExecutionMode(const ExecutionMode executionMode)

{

VerifyExecutionMode(executionMode);

this->executionMode = executionMode;

}

bool FunctionBody::IsInterpreterExecutionMode() const

{

return GetExecutionMode() <= ExecutionMode::ProfilingInterpreter;

}

bool FunctionBody::TryTransitionToNextExecutionMode()

{

Assert(initializedExecutionModeAndLimits);

switch(GetExecutionMode())

{

case ExecutionMode::Interpreter:

if(interpretedCount < interpreterLimit)

{

VerifyExecutionMode(GetExecutionMode());

return false;

}

CommitExecutedIterations(interpreterLimit, interpreterLimit);

goto TransitionToFullJit;

TransitionToAutoProfilingInterpreter:

if(autoProfilingInterpreter0Limit != 0 || autoProfilingInterpreter1Limit != 0)

{

SetExecutionMode(ExecutionMode::AutoProfilingInterpreter);

interpretedCount = 0;

return true;

}

goto TransitionFromAutoProfilingInterpreter;

case ExecutionMode::AutoProfilingInterpreter:

{

uint16 &autoProfilingInterpreterLimit =

autoProfilingInterpreter0Limit == 0 && profilingInterpreter0Limit == 0

? autoProfilingInterpreter1Limit

: autoProfilingInterpreter0Limit;

if(interpretedCount < autoProfilingInterpreterLimit)

{

VerifyExecutionMode(GetExecutionMode());

return false;

}

CommitExecutedIterations(autoProfilingInterpreterLimit, autoProfilingInterpreterLimit);

// fall through

}

TransitionFromAutoProfilingInterpreter:

Assert(autoProfilingInterpreter0Limit == 0 || autoProfilingInterpreter1Limit == 0);

if(profilingInterpreter0Limit == 0 && autoProfilingInterpreter1Limit == 0)

{

goto TransitionToSimpleJit;

}

// fall through

TransitionToProfilingInterpreter:

if(profilingInterpreter0Limit != 0 || profilingInterpreter1Limit != 0)

{

SetExecutionMode(ExecutionMode::ProfilingInterpreter);

interpretedCount = 0;

return true;

}

goto TransitionFromProfilingInterpreter;

case ExecutionMode::ProfilingInterpreter:

{

uint16 &profilingInterpreterLimit =

profilingInterpreter0Limit == 0 && autoProfilingInterpreter1Limit == 0 && simpleJitLimit == 0

? profilingInterpreter1Limit

: profilingInterpreter0Limit;

if(interpretedCount < profilingInterpreterLimit)

{

VerifyExecutionMode(GetExecutionMode());

return false;

}

CommitExecutedIterations(profilingInterpreterLimit, profilingInterpreterLimit);

// fall through

}

TransitionFromProfilingInterpreter:

Assert(profilingInterpreter0Limit == 0 || profilingInterpreter1Limit == 0);

if(autoProfilingInterpreter1Limit == 0 && simpleJitLimit == 0 && profilingInterpreter1Limit == 0)

{

goto TransitionToFullJit;

}

goto TransitionToAutoProfilingInterpreter;

TransitionToSimpleJit:

if(simpleJitLimit != 0)

{

SetExecutionMode(ExecutionMode::SimpleJit);

// Zero the interpreted count here too, so that we can determine how many interpreter iterations ran

// while waiting for simple JIT

interpretedCount = 0;

return true;

}

goto TransitionToProfilingInterpreter;

case ExecutionMode::SimpleJit:

{

FunctionEntryPointInfo \*const simpleJitEntryPointInfo = GetSimpleJitEntryPointInfo();

if(!simpleJitEntryPointInfo || simpleJitEntryPointInfo->callsCount != 0)

{

VerifyExecutionMode(GetExecutionMode());

return false;

}

CommitExecutedIterations(simpleJitLimit, simpleJitLimit);

goto TransitionToProfilingInterpreter;

}

TransitionToFullJit:

if(DoFullJit())

{

SetExecutionMode(ExecutionMode::FullJit);

return true;

}

// fall through

case ExecutionMode::FullJit:

VerifyExecutionMode(GetExecutionMode());

return false;

default:

Assert(false);

\_\_assume(false);

}

}

void FunctionBody::TryTransitionToNextInterpreterExecutionMode()

{

Assert(IsInterpreterExecutionMode());

TryTransitionToNextExecutionMode();

SetExecutionMode(GetInterpreterExecutionMode(false));

}

void FunctionBody::SetIsSpeculativeJitCandidate()

{

// This function is a candidate for speculative JIT. Ensure that it is profiled immediately by transitioning out of the

// auto-profiling interpreter mode.

if(GetExecutionMode() != ExecutionMode::AutoProfilingInterpreter || GetProfiledIterations() != 0)

{

return;

}

TraceExecutionMode("IsSpeculativeJitCandidate (before)");

if(autoProfilingInterpreter0Limit != 0)

{

(profilingInterpreter0Limit == 0 ? profilingInterpreter0Limit : autoProfilingInterpreter1Limit) +=

autoProfilingInterpreter0Limit;

autoProfilingInterpreter0Limit = 0;

}

else if(profilingInterpreter0Limit == 0)

{

profilingInterpreter0Limit += autoProfilingInterpreter1Limit;

autoProfilingInterpreter1Limit = 0;

}

TraceExecutionMode("IsSpeculativeJitCandidate");

TryTransitionToNextInterpreterExecutionMode();

}

bool FunctionBody::TryTransitionToJitExecutionMode()

{

const ExecutionMode previousExecutionMode = GetExecutionMode();

TryTransitionToNextExecutionMode();

switch(GetExecutionMode())

{

case ExecutionMode::SimpleJit:

break;

case ExecutionMode::FullJit:

if(fullJitRequeueThreshold == 0)

{

break;

}

--fullJitRequeueThreshold;

return false;

default:

return false;

}

if(GetExecutionMode() != previousExecutionMode)

{

TraceExecutionMode();

}

return true;

}

void FunctionBody::TransitionToSimpleJitExecutionMode()

{

CommitExecutedIterations();

interpreterLimit = 0;

autoProfilingInterpreter0Limit = 0;

profilingInterpreter0Limit = 0;

autoProfilingInterpreter1Limit = 0;

fullJitThreshold = simpleJitLimit + profilingInterpreter1Limit;

VerifyExecutionModeLimits();

SetExecutionMode(ExecutionMode::SimpleJit);

}

void FunctionBody::TransitionToFullJitExecutionMode()

{

CommitExecutedIterations();

interpreterLimit = 0;

autoProfilingInterpreter0Limit = 0;

profilingInterpreter0Limit = 0;

autoProfilingInterpreter1Limit = 0;

simpleJitLimit = 0;

profilingInterpreter1Limit = 0;

fullJitThreshold = 0;

VerifyExecutionModeLimits();

SetExecutionMode(ExecutionMode::FullJit);

}

void FunctionBody::VerifyExecutionModeLimits()

{

Assert(initializedExecutionModeAndLimits);

Assert(

(

interpreterLimit +

autoProfilingInterpreter0Limit +

profilingInterpreter0Limit +

autoProfilingInterpreter1Limit +

simpleJitLimit +

profilingInterpreter1Limit

) == fullJitThreshold);

}

void FunctionBody::InitializeExecutionModeAndLimits()

{

DebugOnly(initializedExecutionModeAndLimits = true);

const ConfigFlagsTable &configFlags = Configuration::Global.flags;

interpreterLimit = 0;

autoProfilingInterpreter0Limit = static\_cast<uint16>(configFlags.AutoProfilingInterpreter0Limit);

profilingInterpreter0Limit = static\_cast<uint16>(configFlags.ProfilingInterpreter0Limit);

autoProfilingInterpreter1Limit = static\_cast<uint16>(configFlags.AutoProfilingInterpreter1Limit);

simpleJitLimit = static\_cast<uint16>(configFlags.SimpleJitLimit);

profilingInterpreter1Limit = static\_cast<uint16>(configFlags.ProfilingInterpreter1Limit);

// Based on which execution modes are disabled, calculate the number of additional iterations that need to be covered by

// the execution mode that will scale with the full JIT threshold

uint16 scale = 0;

const bool doInterpreterProfile = DoInterpreterProfile();

if(!doInterpreterProfile)

{

scale +=

autoProfilingInterpreter0Limit +

profilingInterpreter0Limit +

autoProfilingInterpreter1Limit +

profilingInterpreter1Limit;

autoProfilingInterpreter0Limit = 0;

profilingInterpreter0Limit = 0;

autoProfilingInterpreter1Limit = 0;

profilingInterpreter1Limit = 0;

}

else if(!DoInterpreterAutoProfile())

{

scale += autoProfilingInterpreter0Limit + autoProfilingInterpreter1Limit;

autoProfilingInterpreter0Limit = 0;

autoProfilingInterpreter1Limit = 0;

if(!IsNewSimpleJit())

{

simpleJitLimit += profilingInterpreter0Limit;

profilingInterpreter0Limit = 0;

}

}

if(!DoSimpleJit())

{

if(!IsNewSimpleJit() && doInterpreterProfile)

{

// The old simple JIT is off, but since it does profiling, it will be replaced with the profiling interpreter

profilingInterpreter1Limit += simpleJitLimit;

}

else

{

scale += simpleJitLimit;

}

simpleJitLimit = 0;

}

if(!DoFullJit())

{

scale += profilingInterpreter1Limit;

profilingInterpreter1Limit = 0;

}

uint16 fullJitThreshold =

static\_cast<uint16>(

configFlags.AutoProfilingInterpreter0Limit +

configFlags.ProfilingInterpreter0Limit +

configFlags.AutoProfilingInterpreter1Limit +

configFlags.SimpleJitLimit +

configFlags.ProfilingInterpreter1Limit);

if(!configFlags.EnforceExecutionModeLimits)

{

/\*

Scale the full JIT threshold based on some heuristics:

- If the % of code in loops is > 50, scale by 1

- Byte-code size of code outside loops

- If the size is < 50, scale by 1.2

- If the size is < 100, scale by 1.4

- If the size is >= 100, scale by 1.6

\*/

const uint loopPercentage = GetByteCodeInLoopCount() \* 100 / max(1u, GetByteCodeCount());

if(loopPercentage <= 50)

{

const uint straightLineSize = GetByteCodeCount() - GetByteCodeInLoopCount();

double fullJitDelayMultiplier;

if(straightLineSize < 50)

{

fullJitDelayMultiplier = 1.2;

}

else if(straightLineSize < 100)

{

fullJitDelayMultiplier = 1.4;

}

else

{

fullJitDelayMultiplier = 1.6;

}

const uint16 newFullJitThreshold = static\_cast<uint16>(fullJitThreshold \* fullJitDelayMultiplier);

scale += newFullJitThreshold - fullJitThreshold;

fullJitThreshold = newFullJitThreshold;

}

}

Assert(fullJitThreshold >= scale);

this->fullJitThreshold = fullJitThreshold - scale;

interpretedCount = 0;

SetExecutionMode(GetDefaultInterpreterExecutionMode());

SetFullJitThreshold(fullJitThreshold);

TryTransitionToNextInterpreterExecutionMode();

}

void FunctionBody::ReinitializeExecutionModeAndLimits()

{

wasCalledFromLoop = false;

fullJitRequeueThreshold = 0;

committedProfiledIterations = 0;

InitializeExecutionModeAndLimits();

}

void FunctionBody::SetFullJitThreshold(const uint16 newFullJitThreshold, const bool skipSimpleJit)

{

Assert(initializedExecutionModeAndLimits);

Assert(GetExecutionMode() != ExecutionMode::FullJit);

int scale = newFullJitThreshold - fullJitThreshold;

if(scale == 0)

{

VerifyExecutionModeLimits();

return;

}

fullJitThreshold = newFullJitThreshold;

const auto ScaleLimit = [&](uint16 &limit) -> bool

{

Assert(scale != 0);

const int limitScale = max(-static\_cast<int>(limit), scale);

const int newLimit = limit + limitScale;

Assert(static\_cast<int>(static\_cast<uint16>(newLimit)) == newLimit);

limit = static\_cast<uint16>(newLimit);

scale -= limitScale;

Assert(limit == 0 || scale == 0);

if(&limit == simpleJitLimit.AddressOf())

{

FunctionEntryPointInfo \*const simpleJitEntryPointInfo = GetSimpleJitEntryPointInfo();

if(GetDefaultFunctionEntryPointInfo() == simpleJitEntryPointInfo)

{

Assert(GetExecutionMode() == ExecutionMode::SimpleJit);

const int newSimpleJitCallCount = max(0, simpleJitEntryPointInfo->callsCount + limitScale);

Assert(static\_cast<int>(static\_cast<uint16>(newSimpleJitCallCount)) == newSimpleJitCallCount);

SetSimpleJitCallCount(static\_cast<uint16>(newSimpleJitCallCount));

}

}

return scale == 0;

};

/\*

Determine which execution mode's limit scales with the full JIT threshold, in order of preference:

- New simple JIT

- Auto-profiling interpreter 1

- Auto-profiling interpreter 0

- Interpreter

- Profiling interpreter 0 (when using old simple JIT)

- Old simple JIT

- Profiling interpreter 1

- Profiling interpreter 0 (when using new simple JIT)

\*/

const bool doSimpleJit = DoSimpleJit();

const bool doInterpreterProfile = DoInterpreterProfile();

const bool fullyScaled =

IsNewSimpleJit() && doSimpleJit && ScaleLimit(simpleJitLimit) ||

(

doInterpreterProfile

? DoInterpreterAutoProfile() &&

(ScaleLimit(autoProfilingInterpreter1Limit) || ScaleLimit(autoProfilingInterpreter0Limit))

: ScaleLimit(interpreterLimit)

) ||

(

IsNewSimpleJit()

? doInterpreterProfile &&

(ScaleLimit(profilingInterpreter1Limit) || ScaleLimit(profilingInterpreter0Limit))

: doInterpreterProfile && ScaleLimit(profilingInterpreter0Limit) ||

doSimpleJit && ScaleLimit(simpleJitLimit) ||

doInterpreterProfile && ScaleLimit(profilingInterpreter1Limit)

);

Assert(fullyScaled);

Assert(scale == 0);

if(GetExecutionMode() != ExecutionMode::SimpleJit)

{

Assert(IsInterpreterExecutionMode());

if(simpleJitLimit != 0 &&

(skipSimpleJit || simpleJitLimit < DEFAULT\_CONFIG\_MinSimpleJitIterations) &&

!PHASE\_FORCE(Phase::SimpleJitPhase, this))

{

// Simple JIT code has not yet been generated, and was either requested to be skipped, or the limit was scaled

// down too much. Skip simple JIT by moving any remaining iterations to an equivalent interpreter execution

// mode.

(IsNewSimpleJit() ? autoProfilingInterpreter1Limit : profilingInterpreter1Limit) += simpleJitLimit;

simpleJitLimit = 0;

TryTransitionToNextInterpreterExecutionMode();

}

}

VerifyExecutionModeLimits();

}

void FunctionBody::CommitExecutedIterations()

{

Assert(initializedExecutionModeAndLimits);

switch(GetExecutionMode())

{

case ExecutionMode::Interpreter:

CommitExecutedIterations(interpreterLimit, interpretedCount);

break;

case ExecutionMode::AutoProfilingInterpreter:

CommitExecutedIterations(

autoProfilingInterpreter0Limit == 0 && profilingInterpreter0Limit == 0

? autoProfilingInterpreter1Limit

: autoProfilingInterpreter0Limit,

interpretedCount);

break;

case ExecutionMode::ProfilingInterpreter:

CommitExecutedIterations(

GetSimpleJitEntryPointInfo()

? profilingInterpreter1Limit

: profilingInterpreter0Limit,

interpretedCount);

break;

case ExecutionMode::SimpleJit:

CommitExecutedIterations(simpleJitLimit, GetSimpleJitExecutedIterations());

break;

case ExecutionMode::FullJit:

break;

default:

Assert(false);

\_\_assume(false);

}

}

void FunctionBody::CommitExecutedIterations(uint16 &limit, const uint executedIterations)

{

Assert(initializedExecutionModeAndLimits);

Assert(

&limit == interpreterLimit.AddressOf() ||

&limit == autoProfilingInterpreter0Limit.AddressOf() ||

&limit == profilingInterpreter0Limit.AddressOf() ||

&limit == autoProfilingInterpreter1Limit.AddressOf() ||

&limit == simpleJitLimit.AddressOf() ||

&limit == profilingInterpreter1Limit.AddressOf());

const uint16 clampedExecutedIterations = executedIterations >= limit ? limit : static\_cast<uint16>(executedIterations);

Assert(fullJitThreshold >= clampedExecutedIterations);

fullJitThreshold -= clampedExecutedIterations;

limit -= clampedExecutedIterations;

VerifyExecutionModeLimits();

if(&limit == profilingInterpreter0Limit.AddressOf() ||

!IsNewSimpleJit() && &limit == simpleJitLimit.AddressOf() ||

&limit == profilingInterpreter1Limit.AddressOf())

{

const uint16 newCommittedProfiledIterations = committedProfiledIterations + clampedExecutedIterations;

committedProfiledIterations =

newCommittedProfiledIterations >= committedProfiledIterations ? newCommittedProfiledIterations : MAXUINT16;

}

}

uint16 FunctionBody::GetSimpleJitExecutedIterations() const

{

Assert(initializedExecutionModeAndLimits);

Assert(GetExecutionMode() == ExecutionMode::SimpleJit);

FunctionEntryPointInfo \*const simpleJitEntryPointInfo = GetSimpleJitEntryPointInfo();

if(!simpleJitEntryPointInfo)

{

return 0;

}

// Simple JIT counts down and transitions on overflow

const uint8 callCount = simpleJitEntryPointInfo->callsCount;

Assert(simpleJitLimit == 0 ? callCount == 0 : simpleJitLimit > callCount);

return callCount == 0 ? simpleJitLimit : simpleJitLimit - callCount - 1;

}

void FunctionBody::ResetSimpleJitLimitAndCallCount()

{

Assert(initializedExecutionModeAndLimits);

Assert(GetExecutionMode() == ExecutionMode::SimpleJit);

Assert(GetDefaultFunctionEntryPointInfo() == GetSimpleJitEntryPointInfo());

const uint16 simpleJitNewLimit = static\_cast<uint8>(Configuration::Global.flags.SimpleJitLimit);

Assert(simpleJitNewLimit == Configuration::Global.flags.SimpleJitLimit);

if(simpleJitLimit < simpleJitNewLimit)

{

fullJitThreshold += simpleJitNewLimit - simpleJitLimit;

simpleJitLimit = simpleJitNewLimit;

}

interpretedCount = 0;

ResetSimpleJitCallCount();

}

void FunctionBody::SetSimpleJitCallCount(const uint16 simpleJitLimit) const

{

Assert(GetExecutionMode() == ExecutionMode::SimpleJit);

Assert(GetDefaultFunctionEntryPointInfo() == GetSimpleJitEntryPointInfo());

// Simple JIT counts down and transitions on overflow

const uint8 limit = static\_cast<uint8>(min(0xffui16, simpleJitLimit));

GetSimpleJitEntryPointInfo()->callsCount = limit == 0 ? 0 : limit - 1;

}

void FunctionBody::ResetSimpleJitCallCount()

{

SetSimpleJitCallCount(

simpleJitLimit > interpretedCount

? simpleJitLimit - static\_cast<uint16>(interpretedCount)

: 0ui16);

}

uint16 FunctionBody::GetProfiledIterations() const

{

Assert(initializedExecutionModeAndLimits);

uint16 profiledIterations = committedProfiledIterations;

switch(GetExecutionMode())

{

case ExecutionMode::ProfilingInterpreter:

{

const uint16 clampedInterpretedCount =

interpretedCount <= MAXUINT16

? static\_cast<uint16>(interpretedCount)

: MAXUINT16;

const uint16 newProfiledIterations = profiledIterations + clampedInterpretedCount;

profiledIterations = newProfiledIterations >= profiledIterations ? newProfiledIterations : MAXUINT16;

break;

}

case ExecutionMode::SimpleJit:

if(!IsNewSimpleJit())

{

const uint16 newProfiledIterations = profiledIterations + GetSimpleJitExecutedIterations();

profiledIterations = newProfiledIterations >= profiledIterations ? newProfiledIterations : MAXUINT16;

}

break;

}

return profiledIterations;

}

void FunctionBody::OnFullJitDequeued(const FunctionEntryPointInfo \*const entryPointInfo)

{

Assert(initializedExecutionModeAndLimits);

Assert(GetExecutionMode() == ExecutionMode::FullJit);

Assert(entryPointInfo);

if(entryPointInfo != GetDefaultFunctionEntryPointInfo())

{

return;

}

// Re-queue the full JIT work item after this many iterations

fullJitRequeueThreshold = static\_cast<uint16>(DEFAULT\_CONFIG\_FullJitRequeueThreshold);

}

void FunctionBody::TraceExecutionMode(const char \*const eventDescription) const

{

Assert(initializedExecutionModeAndLimits);

if(PHASE\_TRACE(Phase::ExecutionModePhase, this))

{

DoTraceExecutionMode(eventDescription);

}

}

void FunctionBody::TraceInterpreterExecutionMode() const

{

Assert(initializedExecutionModeAndLimits);

if(!PHASE\_TRACE(Phase::ExecutionModePhase, this))

{

return;

}

switch(GetExecutionMode())

{

case ExecutionMode::Interpreter:

case ExecutionMode::AutoProfilingInterpreter:

case ExecutionMode::ProfilingInterpreter:

DoTraceExecutionMode(nullptr);

break;

}

}

void FunctionBody::DoTraceExecutionMode(const char \*const eventDescription) const

{

Assert(PHASE\_TRACE(Phase::ExecutionModePhase, this));

Assert(initializedExecutionModeAndLimits);

wchar\_t functionIdString[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(

L"ExecutionMode - "

L"function: %s (%s), "

L"mode: %S, "

L"size: %u, "

L"limits: %hu.%hu.%hu.%hu.%hu = %hu",

GetDisplayName(),

GetDebugNumberSet(functionIdString),

ExecutionModeName(executionMode),

GetByteCodeCount(),

interpreterLimit + autoProfilingInterpreter0Limit,

profilingInterpreter0Limit,

autoProfilingInterpreter1Limit,

simpleJitLimit,

profilingInterpreter1Limit,

fullJitThreshold);

if(eventDescription)

{

Output::Print(L", event: %S", eventDescription);

}

Output::Print(L"\n");

Output::Flush();

}

bool FunctionBody::IsNewSimpleJit()

{

return CONFIG\_FLAG(NewSimpleJit);

}

bool FunctionBody::DoSimpleJit() const

{

return

!PHASE\_OFF(Js::SimpleJitPhase, this) &&

!GetScriptContext()->GetConfig()->IsNoNative() &&

!GetScriptContext()->IsInDebugMode() &&

DoInterpreterProfile() &&

(!IsNewSimpleJit() || DoInterpreterAutoProfile()) &&

!IsGenerator(); // Generator JIT requires bailout which SimpleJit cannot do since it skips GlobOpt

}

bool FunctionBody::DoSimpleJitDynamicProfile() const

{

Assert(DoSimpleJit());

return !PHASE\_OFF(Js::SimpleJitDynamicProfilePhase, this) && !IsNewSimpleJit();

}

bool FunctionBody::DoInterpreterProfile() const

{

#if ENABLE\_PROFILE\_INFO

// Switch off profiling is asmJsFunction

if (this->GetIsAsmJsFunction())

{

return false;

}

else

{

return !PHASE\_OFF(InterpreterProfilePhase, this) && DynamicProfileInfo::IsEnabled(this);

}

#else

return false;

#endif

}

bool FunctionBody::DoInterpreterAutoProfile() const

{

Assert(DoInterpreterProfile());

return !PHASE\_OFF(InterpreterAutoProfilePhase, this) && !GetScriptContext()->IsInDebugMode();

}

bool FunctionBody::WasCalledFromLoop() const

{

return wasCalledFromLoop;

}

void FunctionBody::SetWasCalledFromLoop()

{

if(wasCalledFromLoop)

{

return;

}

wasCalledFromLoop = true;

if(Configuration::Global.flags.EnforceExecutionModeLimits)

{

if(PHASE\_TRACE(Phase::ExecutionModePhase, this))

{

CommitExecutedIterations();

TraceExecutionMode("WasCalledFromLoop (before)");

}

}

else

{

// This function is likely going to be called frequently since it's called from a loop. Reduce the full JIT

// threshold to realize the full JIT perf benefit sooner.

CommitExecutedIterations();

TraceExecutionMode("WasCalledFromLoop (before)");

if(fullJitThreshold > 1)

{

SetFullJitThreshold(fullJitThreshold / 2, !IsNewSimpleJit());

}

}

{

// Reduce the loop interpreter limit too, for the same reasons as above

const uint oldLoopInterpreterLimit = loopInterpreterLimit;

const uint newLoopInterpreterLimit = GetReducedLoopInterpretCount();

Assert(newLoopInterpreterLimit <= oldLoopInterpreterLimit);

loopInterpreterLimit = newLoopInterpreterLimit;

// Adjust loop headers' interpret counts to ensure that loops will still be profiled a number of times before

// loop bodies are jitted

const uint oldLoopProfileThreshold = GetLoopProfileThreshold(oldLoopInterpreterLimit);

const uint newLoopProfileThreshold = GetLoopProfileThreshold(newLoopInterpreterLimit);

MapLoopHeaders([=](const uint index, LoopHeader \*const loopHeader)

{

const uint interpretedCount = loopHeader->interpretCount;

if(interpretedCount <= newLoopProfileThreshold || interpretedCount >= oldLoopInterpreterLimit)

{

// The loop hasn't been profiled yet and wouldn't have started profiling even with the new profile

// threshold, or it has already been profiled the necessary minimum number of times based on the old limit

return;

}

if(interpretedCount <= oldLoopProfileThreshold)

{

// The loop hasn't been profiled yet, but would have started profiling with the new profile threshold. Start

// profiling on the next iteration.

loopHeader->interpretCount = newLoopProfileThreshold;

return;

}

// The loop has been profiled some already. Preserve the number of profiled iterations.

loopHeader->interpretCount = newLoopProfileThreshold + (interpretedCount - oldLoopProfileThreshold);

});

}

TraceExecutionMode("WasCalledFromLoop");

}

bool FunctionBody::RecentlyBailedOutOfJittedLoopBody() const

{

return recentlyBailedOutOfJittedLoopBody;

}

void FunctionBody::SetRecentlyBailedOutOfJittedLoopBody(const bool value)

{

recentlyBailedOutOfJittedLoopBody = value;

}

uint16 FunctionBody::GetMinProfileIterations()

{

return

static\_cast<uint>(

IsNewSimpleJit()

? DEFAULT\_CONFIG\_MinProfileIterations

: DEFAULT\_CONFIG\_MinProfileIterations\_OldSimpleJit);

}

uint16 FunctionBody::GetMinFunctionProfileIterations()

{

return GetMinProfileIterations();

}

uint FunctionBody::GetMinLoopProfileIterations(const uint loopInterpreterLimit)

{

return min(static\_cast<uint>(GetMinProfileIterations()), loopInterpreterLimit);

}

uint FunctionBody::GetLoopProfileThreshold(const uint loopInterpreterLimit) const

{

return

DoInterpreterProfile()

? DoInterpreterAutoProfile()

? loopInterpreterLimit - GetMinLoopProfileIterations(loopInterpreterLimit)

: 0

: static\_cast<uint>(-1);

}

uint FunctionBody::GetReducedLoopInterpretCount()

{

const uint loopInterpretCount = CONFIG\_FLAG(LoopInterpretCount);

if(CONFIG\_ISENABLED(LoopInterpretCountFlag))

{

return loopInterpretCount;

}

return max(loopInterpretCount / 3, GetMinLoopProfileIterations(loopInterpretCount));

}

uint FunctionBody::GetLoopInterpretCount(LoopHeader\* loopHeader) const

{

if(loopHeader->isNested)

{

Assert(loopInterpreterLimit >= GetReducedLoopInterpretCount());

return GetReducedLoopInterpretCount();

}

return loopInterpreterLimit;

}

bool FunctionBody::DoObjectHeaderInlining()

{

return !PHASE\_OFF1(ObjectHeaderInliningPhase);

}

bool FunctionBody::DoObjectHeaderInliningForConstructors()

{

return !PHASE\_OFF1(ObjectHeaderInliningForConstructorsPhase) && DoObjectHeaderInlining();

}

bool FunctionBody::DoObjectHeaderInliningForConstructor(const uint32 inlineSlotCapacity)

{

return inlineSlotCapacity == 0 ? DoObjectHeaderInliningForEmptyObjects() : DoObjectHeaderInliningForConstructors();

}

bool FunctionBody::DoObjectHeaderInliningForObjectLiterals()

{

return !PHASE\_OFF1(ObjectHeaderInliningForObjectLiteralsPhase) && DoObjectHeaderInlining();

}

bool FunctionBody::DoObjectHeaderInliningForObjectLiteral(const uint32 inlineSlotCapacity)

{

return

inlineSlotCapacity == 0

? DoObjectHeaderInliningForEmptyObjects()

: DoObjectHeaderInliningForObjectLiterals() &&

inlineSlotCapacity <= static\_cast<uint32>(MaxPreInitializedObjectHeaderInlinedTypeInlineSlotCount);

}

bool FunctionBody::DoObjectHeaderInliningForObjectLiteral(

const PropertyIdArray \*const propIds,

ScriptContext \*const scriptContext)

{

Assert(propIds);

Assert(scriptContext);

return

DoObjectHeaderInliningForObjectLiteral(propIds->count) &&

PathTypeHandlerBase::UsePathTypeHandlerForObjectLiteral(propIds, scriptContext);

}

bool FunctionBody::DoObjectHeaderInliningForEmptyObjects()

{

#pragma prefast(suppress:6237, "(<zero> && <expression>) is always zero. <expression> is never evaluated and might have side effects.")

return PHASE\_ON1(ObjectHeaderInliningForEmptyObjectsPhase) && DoObjectHeaderInlining();

}

void FunctionBody::Finalize(bool isShutdown)

{

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.Instrument.IsEnabled(Js::LinearScanPhase, this->GetSourceContextId(), this->GetLocalFunctionId()))

{

this->DumpRegStats(this);

}

#endif

this->Cleanup(isShutdown);

this->CleanupSourceInfo(isShutdown);

this->ClearNestedFunctionParentFunctionReference();

this->CleanupFunctionProxyCounters();

}

void FunctionBody::CleanupSourceInfo(bool isScriptContextClosing)

{

Assert(this->cleanedUp);

if (!sourceInfoCleanedUp)

{

if (GetIsFuncRegistered() && !isScriptContextClosing)

{

// If our function is registered, then there must

// be a Utf8SourceInfo pinned by it.

Assert(this->m\_utf8SourceInfo);

this->m\_utf8SourceInfo->RemoveFunctionBody(this);

}

if (this->m\_sourceInfo.pSpanSequence != nullptr)

{

HeapDelete(this->m\_sourceInfo.pSpanSequence);

this->m\_sourceInfo.pSpanSequence = nullptr;

}

sourceInfoCleanedUp = true;

}

}

template<bool IsScriptContextShutdown>

void FunctionBody::CleanUpInlineCaches()

{

uint unregisteredInlineCacheCount = 0;

if (nullptr != this->inlineCaches)

{

// Inline caches are in this order

// plain inline cache

// root object load inline cache

// root object store inline cache

// isInst inline cache

// The inlineCacheCount includes all but isInst inline cache

uint i = 0;

uint plainInlineCacheEnd = GetRootObjectLoadInlineCacheStart();

for (; i < plainInlineCacheEnd; i++)

{

if (nullptr != this->inlineCaches[i])

{

InlineCache\* inlineCache = (InlineCache\*)this->inlineCaches[i];

if (IsScriptContextShutdown)

{

memset(inlineCache, 0, sizeof(InlineCache));

}

else

{

if (inlineCache->RemoveFromInvalidationList())

{

unregisteredInlineCacheCount++;

}

AllocatorDelete(InlineCacheAllocator, this->m\_scriptContext->GetInlineCacheAllocator(), inlineCache);

}

}

}

RootObjectBase \* rootObjectBase = this->GetRootObject();

uint rootObjectLoadInlineCacheEnd = GetRootObjectLoadMethodInlineCacheStart();

for (; i < rootObjectLoadInlineCacheEnd; i++)

{

if (nullptr != this->inlineCaches[i])

{

InlineCache\* inlineCache = (InlineCache\*)this->inlineCaches[i];

if (IsScriptContextShutdown)

{

memset(inlineCache, 0, sizeof(InlineCache));

}

else

{

// A single root object inline caches for a given property is shared by all functions. It is ref counted

// and doesn't get released to the allocator until there are no more outstanding references. Thus we don't need

// to (and, in fact, cannot) remove it from the invalidation list here. Instead, we'll do it in ReleaseInlineCache

// when there are no more outstanding references.

rootObjectBase->ReleaseInlineCache(this->GetPropertyIdFromCacheId(i), false, false, IsScriptContextShutdown);

}

}

}

uint rootObjectLoadMethodInlineCacheEnd = GetRootObjectStoreInlineCacheStart();

for (; i < rootObjectLoadMethodInlineCacheEnd; i++)

{

if (nullptr != this->inlineCaches[i])

{

InlineCache\* inlineCache = (InlineCache\*)this->inlineCaches[i];

if (IsScriptContextShutdown)

{

memset(inlineCache, 0, sizeof(InlineCache));

}

else

{

// A single root object inline caches for a given property is shared by all functions. It is ref counted

// and doesn't get released to the allocator until there are no more outstanding references. Thus we don't need

// to (and, in fact, cannot) remove it from the invalidation list here. Instead, we'll do it in ReleaseInlineCache

// when there are no more outstanding references.

rootObjectBase->ReleaseInlineCache(this->GetPropertyIdFromCacheId(i), true, false, IsScriptContextShutdown);

}

}

}

uint rootObjectStoreInlineCacheEnd = this->GetInlineCacheCount();

for (; i < rootObjectStoreInlineCacheEnd; i++)

{

if (nullptr != this->inlineCaches[i])

{

InlineCache\* inlineCache = (InlineCache\*)this->inlineCaches[i];

if (IsScriptContextShutdown)

{

memset(inlineCache, 0, sizeof(InlineCache));

}

else

{

// A single root object inline caches for a given property is shared by all functions. It is ref counted

// and doesn't get released to the allocator until there are no more outstanding references. Thus we don't need

// to (and, in fact, cannot) remove it from the invalidation list here. Instead, we'll do it in ReleaseInlineCache

// when there are no more outstanding references.

rootObjectBase->ReleaseInlineCache(this->GetPropertyIdFromCacheId(i), false, true, IsScriptContextShutdown);

}

}

}

uint totalCacheCount = inlineCacheCount + GetIsInstInlineCacheCount();

for (; i < totalCacheCount; i++)

{

if (nullptr != this->inlineCaches[i])

{

IsInstInlineCache\* inlineCache = (IsInstInlineCache\*)this->inlineCaches[i];

if (IsScriptContextShutdown)

{

memset(inlineCache, 0, sizeof(IsInstInlineCache));

}

else

{

AllocatorDelete(IsInstInlineCacheAllocator, this->m\_scriptContext->GetIsInstInlineCacheAllocator(), inlineCache);

}

}

}

this->inlineCaches = nullptr;

}

if (nullptr != this->m\_codeGenRuntimeData)

{

for (ProfileId i = 0; i < this->profiledCallSiteCount; i++)

{

const FunctionCodeGenRuntimeData\* runtimeData = this->m\_codeGenRuntimeData[i];

if (nullptr != runtimeData)

{

runtimeData->MapInlineCaches([&](InlineCache\* inlineCache)

{

if (nullptr != inlineCache)

{

if (IsScriptContextShutdown)

{

memset(inlineCache, 0, sizeof(InlineCache));

}

else

{

if (inlineCache->RemoveFromInvalidationList())

{

unregisteredInlineCacheCount++;

}

AllocatorDelete(InlineCacheAllocator, this->m\_scriptContext->GetInlineCacheAllocator(), inlineCache);

}

}

});

}

}

}

if (nullptr != this->m\_codeGenGetSetRuntimeData)

{

for (uint i = 0; i < this->GetInlineCacheCount(); i++)

{

const FunctionCodeGenRuntimeData\* runtimeData = this->m\_codeGenGetSetRuntimeData[i];

if (nullptr != runtimeData)

{

runtimeData->MapInlineCaches([&](InlineCache\* inlineCache)

{

if (nullptr != inlineCache)

{

if (IsScriptContextShutdown)

{

memset(inlineCache, 0, sizeof(InlineCache));

}

else

{

if (inlineCache->RemoveFromInvalidationList())

{

unregisteredInlineCacheCount++;

}

AllocatorDelete(InlineCacheAllocator, this->m\_scriptContext->GetInlineCacheAllocator(), inlineCache);

}

}

});

}

}

}

if (!IsScriptContextShutdown)

{

ThreadContext\* threadContext = this->m\_scriptContext->GetThreadContext();

if (unregisteredInlineCacheCount > 0)

{

threadContext->NotifyInlineCacheBatchUnregistered(unregisteredInlineCacheCount);

}

}

while (polymorphicInlineCachesHead)

{

polymorphicInlineCachesHead->Finalize(IsScriptContextShutdown);

}

polymorphicInlineCaches.Reset();

}

void FunctionBody::CleanupRecyclerData(bool isShutdown, bool doEntryPointCleanupCaptureStack)

{

// If we're not shutting down (i.e 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

// (i.e it's 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)

{

CleanUpInlineCaches<true>();

}

else

{

CleanUpInlineCaches<false>();

}

if (this->entryPoints)

{

#if defined(ENABLE\_DEBUG\_CONFIG\_OPTIONS) && !(DBG)

// On fretest builds, capture the stack only if the FreTestDiagMode switch is on

doEntryPointCleanupCaptureStack = doEntryPointCleanupCaptureStack && Js::Configuration::Global.flags.FreTestDiagMode;

#endif

this->MapEntryPoints([=](int index, FunctionEntryPointInfo\* entryPoint)

{

if (nullptr != entryPoint)

{

// Finalize = Free up work item if it hasn't been released yet + entry point clean up

// isShutdown is false because cleanup is called only in the !isShutdown case

entryPoint->Finalize(isShutdown);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

// Do this separately since calling EntryPoint::Finalize doesn't capture the stack trace

// and in some calls to CleanupRecyclerData, we do want the stack trace captured.

if (doEntryPointCleanupCaptureStack)

{

entryPoint->CaptureCleanupStackTrace();

}

#endif

}

});

this->MapLoopHeaders([=](uint loopNumber, LoopHeader\* header)

{

bool shuttingDown = isShutdown;

header->MapEntryPoints([=](int index, LoopEntryPointInfo\* entryPoint)

{

entryPoint->Cleanup(shuttingDown, doEntryPointCleanupCaptureStack);

});

});

}

#ifdef PERF\_COUNTERS

this->CleanupPerfCounter();

#endif

}

//

// Removes all references of the function body and causes clean up of entry points.

// If the cleanup has already occurred before this would be a no-op.

//

void FunctionBody::Cleanup(bool isScriptContextClosing)

{

if (cleanedUp)

{

return;

}

CleanupRecyclerData(isScriptContextClosing, false /\* capture entry point cleanup stack trace \*/);

this->ResetObjectLiteralTypes();

// Manually clear these values to break any circular references

// that might prevent the script context from being disposed

this->auxBlock = nullptr;

this->auxContextBlock = nullptr;

this->byteCodeBlock = nullptr;

this->entryPoints = nullptr;

this->loopHeaderArray = nullptr;

this->m\_constTable = nullptr;

this->m\_codeGenRuntimeData = nullptr;

this->m\_codeGenGetSetRuntimeData = nullptr;

this->inlineCaches = nullptr;

this->polymorphicInlineCaches.Reset();

this->polymorphicInlineCachesHead = nullptr;

this->cacheIdToPropertyIdMap = nullptr;

this->referencedPropertyIdMap = nullptr;

this->literalRegexes = nullptr;

this->propertyIdsForScopeSlotArray = nullptr;

this->propertyIdOnRegSlotsContainer = nullptr;

#if DYNAMIC\_INTERPRETER\_THUNK

if (this->HasInterpreterThunkGenerated())

{

JS\_ETW(EtwTrace::LogMethodInterpreterThunkUnloadEvent(this));

if (!isScriptContextClosing)

{

if (m\_isAsmJsFunction)

{

m\_scriptContext->ReleaseDynamicAsmJsInterpreterThunk((BYTE\*)this->m\_dynamicInterpreterThunk, /\*addtoFreeList\*/!isScriptContextClosing);

}

else

{

m\_scriptContext->ReleaseDynamicInterpreterThunk((BYTE\*)this->m\_dynamicInterpreterThunk, /\*addtoFreeList\*/!isScriptContextClosing);

}

}

}

#endif

#if ENABLE\_PROFILE\_INFO

this->polymorphicCallSiteInfoHead = nullptr;

#endif

this->cleanedUp = true;

}

#ifdef PERF\_COUNTERS

void FunctionBody::CleanupPerfCounter()

{

// We might not have the byte code block yet if we defer parsed.

DWORD byteCodeSize = (this->byteCodeBlock? this->byteCodeBlock->GetLength() : 0)

+ (this->auxBlock? this->auxBlock->GetLength() : 0)

+ (this->auxContextBlock? this->auxContextBlock->GetLength() : 0);

PERF\_COUNTER\_SUB(Code, DynamicByteCodeSize, byteCodeSize);

if (this->m\_isDeserializedFunction)

{

PERF\_COUNTER\_DEC(Code, DeserializedFunctionBody);

}

PERF\_COUNTER\_SUB(Code, TotalByteCodeSize, byteCodeSize);

}

#endif

void FunctionBody::CaptureDynamicProfileState(FunctionEntryPointInfo\* entryPointInfo)

{

// DisableJIT-TODO: Move this to be under if DYNAMIC\_PROFILE

#if ENABLE\_NATIVE\_CODEGEN

// (See also the FunctionBody member written in CaptureDymamicProfileState.)

this->savedPolymorphicCacheState = entryPointInfo->GetPendingPolymorphicCacheState();

this->savedInlinerVersion = entryPointInfo->GetPendingInlinerVersion();

this->savedImplicitCallsFlags = entryPointInfo->GetPendingImplicitCallFlags();

#endif

}

#if ENABLE\_NATIVE\_CODEGEN

BYTE FunctionBody::GetSavedInlinerVersion() const

{

Assert(this->dynamicProfileInfo != nullptr);

return this->savedInlinerVersion;

}

uint32 FunctionBody::GetSavedPolymorphicCacheState() const

{

Assert(this->dynamicProfileInfo != nullptr);

return this->savedPolymorphicCacheState;

}

#endif

void FunctionBody::SetHasHotLoop()

{

if(hasHotLoop)

{

return;

}

hasHotLoop = true;

if(Configuration::Global.flags.EnforceExecutionModeLimits)

{

return;

}

CommitExecutedIterations();

TraceExecutionMode("HasHotLoop (before)");

if(fullJitThreshold > 1)

{

SetFullJitThreshold(1, true);

}

TraceExecutionMode("HasHotLoop");

}

bool FunctionBody::IsInlineApplyDisabled()

{

return this->disableInlineApply;

}

void FunctionBody::SetDisableInlineApply(bool set)

{

this->disableInlineApply = set;

}

void FunctionBody::InitDisableInlineApply()

{

SetDisableInlineApply(this->functionId != Js::Constants::NoFunctionId && PHASE\_OFF(Js::InlinePhase, this) || PHASE\_OFF(Js::InlineApplyPhase, this));

}

bool FunctionBody::CheckCalleeContextForInlining(FunctionProxy\* calleeFunctionProxy)

{

if (this->GetScriptContext() == calleeFunctionProxy->GetScriptContext())

{

if (this->GetHostSourceContext() == calleeFunctionProxy->GetHostSourceContext() &&

this->GetSecondaryHostSourceContext() == calleeFunctionProxy->GetSecondaryHostSourceContext())

{

return true;

}

}

return false;

}

#if ENABLE\_NATIVE\_CODEGEN

ImplicitCallFlags FunctionBody::GetSavedImplicitCallsFlags() const

{

Assert(this->dynamicProfileInfo != nullptr);

return this->savedImplicitCallsFlags;

}

bool FunctionBody::HasNonBuiltInCallee()

{

for (ProfileId i = 0; i < profiledCallSiteCount; i++)

{

Assert(HasDynamicProfileInfo());

bool ctor;

bool isPolymorphic;

FunctionInfo \*info = dynamicProfileInfo->GetCallSiteInfo(this, i, &ctor, &isPolymorphic);

if (info == nullptr || info->HasBody())

{

return true;

}

}

return false;

}

#endif

void FunctionBody::CheckAndRegisterFuncToDiag(ScriptContext \*scriptContext)

{

// We will register function if, this is not host managed and it was not registered before.

if (GetHostSourceContext() == Js::Constants::NoHostSourceContext

&& !m\_isFuncRegisteredToDiag

&& !scriptContext->GetDebugContext()->GetProbeContainer()->IsContextRegistered(GetSecondaryHostSourceContext()))

{

FunctionBody \*pFunc = scriptContext->GetDebugContext()->GetProbeContainer()->GetGlobalFunc(scriptContext, GetSecondaryHostSourceContext());

if (pFunc)

{

// Existing behavior here is to ignore the OOM and since RegisterFuncToDiag

// can throw now, we simply ignore the OOM here

try

{

// Register the function to the PDM as eval code (the debugger app will show file as 'eval code')

pFunc->RegisterFuncToDiag(scriptContext, Constants::EvalCode);

}

catch (Js::OutOfMemoryException)

{

}

scriptContext->GetDebugContext()->GetProbeContainer()->RegisterContextToDiag(GetSecondaryHostSourceContext(), scriptContext->AllocatorForDiagnostics());

m\_isFuncRegisteredToDiag = true;

}

}

else

{

m\_isFuncRegisteredToDiag = true;

}

}

DebuggerScope\* FunctionBody::RecordStartScopeObject(DiagExtraScopesType scopeType, int start, RegSlot scopeLocation, int\* index)

{

Recycler\* recycler = m\_scriptContext->GetRecycler();

if (!GetScopeObjectChain())

{

SetScopeObjectChain(RecyclerNew(recycler, ScopeObjectChain, recycler));

}

// Check if we need to create the scope object or if it already exists from a previous bytecode

// generator pass.

DebuggerScope\* debuggerScope = nullptr;

int currentDebuggerScopeIndex = this->GetNextDebuggerScopeIndex();

if (!this->TryGetDebuggerScopeAt(currentDebuggerScopeIndex, debuggerScope))

{

// Create a new debugger scope.

debuggerScope = AddScopeObject(scopeType, start, scopeLocation);

}

else

{

debuggerScope->UpdateDueToByteCodeRegeneration(scopeType, start, scopeLocation);

}

if(index)

{

\*index = currentDebuggerScopeIndex;

}

return debuggerScope;

}

void FunctionBody::RecordEndScopeObject(DebuggerScope\* currentScope, int end)

{

AssertMsg(currentScope, "No current debugger scope passed in.");

currentScope->SetEnd(end);

}

DebuggerScope \* FunctionBody::AddScopeObject(DiagExtraScopesType scopeType, int start, RegSlot scopeLocation)

{

Assert(GetScopeObjectChain());

DebuggerScope \*scopeObject = RecyclerNew(m\_scriptContext->GetRecycler(), DebuggerScope, m\_scriptContext->GetRecycler(), scopeType, scopeLocation, start);

GetScopeObjectChain()->pScopeChain->Add(scopeObject);

return scopeObject;

}

// Tries to retrieve the debugger scope at the specified index. If the index is out of range, nullptr

// is returned.

bool FunctionBody::TryGetDebuggerScopeAt(int index, DebuggerScope\*& debuggerScope)

{

AssertMsg(this->GetScopeObjectChain(), "TryGetDebuggerScopeAt should only be called with a valid scope chain in place.");

Assert(index >= 0);

const Js::ScopeObjectChain::ScopeObjectChainList\* scopeChain = this->GetScopeObjectChain()->pScopeChain;

if (index < scopeChain->Count())

{

debuggerScope = scopeChain->Item(index);

return true;

}

return false;

}

#if DYNAMIC\_INTERPRETER\_THUNK

DWORD FunctionBody::GetDynamicInterpreterThunkSize() const

{

return InterpreterThunkEmitter::ThunkSize;

}

#endif

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

void

FunctionBody::DumpFullFunctionName()

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"Function %s (%s)", this->GetDisplayName(), this->GetDebugNumberSet(debugStringBuffer));

}

void FunctionBody::DumpFunctionId(bool pad)

{

uint sourceContextId = this->GetSourceContextInfo()->sourceContextId;

if (sourceContextId == Js::Constants::NoSourceContext)

{

if (this->IsDynamicScript())

{

Output::Print(pad? L"Dy.%-3d" : L"Dyn#%d", this->GetLocalFunctionId());

}

else

{

// Function from LoadFile

Output::Print(pad? L"%-5d" : L"#%d", this->GetLocalFunctionId());

}

}

else

{

Output::Print(pad? L"%2d.%-3d" : L"#%d.%d", sourceContextId, this->GetLocalFunctionId());

}

}

#endif

void FunctionBody::EnsureAuxStatementData()

{

if (m\_sourceInfo.m\_auxStatementData == nullptr)

{

Recycler\* recycler = m\_scriptContext->GetRecycler();

// Note: allocating must be consistent with clean up in CleanupToReparse.

m\_sourceInfo.m\_auxStatementData = RecyclerNew(recycler, AuxStatementData);

}

}

/\*static\*/

void FunctionBody::GetShortNameFromUrl(\_\_in LPCWSTR pchUrl, \_Out\_writes\_z\_(cchBuffer) LPWSTR pchShortName, \_\_in size\_t cchBuffer)

{

LPCWSTR pchFile = wcsrchr(pchUrl, L'/');

if (pchFile == nullptr)

{

pchFile = wcsrchr(pchUrl, L'\\');

}

LPCWSTR pchToCopy = pchUrl;

if (pchFile != nullptr)

{

pchToCopy = pchFile + 1;

}

wcscpy\_s(pchShortName, cchBuffer, pchToCopy);

}

FunctionBody::StatementAdjustmentRecordList\* FunctionBody::GetStatementAdjustmentRecords()

{

if (m\_sourceInfo.m\_auxStatementData)

{

return m\_sourceInfo.m\_auxStatementData->m\_statementAdjustmentRecords;

}

return nullptr;

}

FunctionBody::CrossFrameEntryExitRecordList\* FunctionBody::GetCrossFrameEntryExitRecords()

{

if (m\_sourceInfo.m\_auxStatementData)

{

return m\_sourceInfo.m\_auxStatementData->m\_crossFrameBlockEntryExisRecords;

}

return nullptr;

}

void FunctionBody::RecordCrossFrameEntryExitRecord(uint byteCodeOffset, bool isEnterBlock)

{

this->EnsureAuxStatementData();

Recycler\* recycler = this->m\_scriptContext->GetRecycler();

if (this->GetCrossFrameEntryExitRecords() == nullptr)

{

m\_sourceInfo.m\_auxStatementData->m\_crossFrameBlockEntryExisRecords = RecyclerNew(recycler, CrossFrameEntryExitRecordList, recycler);

}

Assert(this->GetCrossFrameEntryExitRecords());

CrossFrameEntryExitRecord record(byteCodeOffset, isEnterBlock);

this->GetCrossFrameEntryExitRecords()->Add(record); // Will copy stack value and put the copy into the container.

}

FunctionBody::AuxStatementData::AuxStatementData() : m\_statementAdjustmentRecords(nullptr), m\_crossFrameBlockEntryExisRecords(nullptr)

{

}

FunctionBody::StatementAdjustmentRecord::StatementAdjustmentRecord() :

m\_byteCodeOffset((uint)Constants::InvalidOffset), m\_adjustmentType(SAT\_None)

{

}

FunctionBody::StatementAdjustmentRecord::StatementAdjustmentRecord(StatementAdjustmentType type, int byteCodeOffset) :

m\_adjustmentType(type), m\_byteCodeOffset(byteCodeOffset)

{

Assert(SAT\_None <= type && type <= SAT\_All);

}

FunctionBody::StatementAdjustmentRecord::StatementAdjustmentRecord(const StatementAdjustmentRecord& other) :

m\_byteCodeOffset(other.m\_byteCodeOffset), m\_adjustmentType(other.m\_adjustmentType)

{

}

uint FunctionBody::StatementAdjustmentRecord::GetByteCodeOffset()

{

Assert(m\_byteCodeOffset != Constants::InvalidOffset);

return m\_byteCodeOffset;

}

FunctionBody::StatementAdjustmentType FunctionBody::StatementAdjustmentRecord::GetAdjustmentType()

{

Assert(this->m\_adjustmentType != SAT\_None);

return m\_adjustmentType;

}

FunctionBody::CrossFrameEntryExitRecord::CrossFrameEntryExitRecord() :

m\_byteCodeOffset((uint)Constants::InvalidOffset), m\_isEnterBlock(false)

{

}

FunctionBody::CrossFrameEntryExitRecord::CrossFrameEntryExitRecord(uint byteCodeOffset, bool isEnterBlock) :

m\_byteCodeOffset(byteCodeOffset), m\_isEnterBlock(isEnterBlock)

{

}

FunctionBody::CrossFrameEntryExitRecord::CrossFrameEntryExitRecord(const CrossFrameEntryExitRecord& other) :

m\_byteCodeOffset(other.m\_byteCodeOffset), m\_isEnterBlock(other.m\_isEnterBlock)

{

}

uint FunctionBody::CrossFrameEntryExitRecord::GetByteCodeOffset() const

{

Assert(m\_byteCodeOffset != Constants::InvalidOffset);

return m\_byteCodeOffset;

}

bool FunctionBody::CrossFrameEntryExitRecord::GetIsEnterBlock()

{

return m\_isEnterBlock;

}

PolymorphicInlineCacheInfo \* EntryPointPolymorphicInlineCacheInfo::GetInlineeInfo(FunctionBody \* inlineeFunctionBody)

{

SListBase<PolymorphicInlineCacheInfo\*>::Iterator iter(&inlineeInfo);

while (iter.Next())

{

PolymorphicInlineCacheInfo \* info = iter.Data();

if (info->GetFunctionBody() == inlineeFunctionBody)

{

return info;

}

}

return nullptr;

}

PolymorphicInlineCacheInfo \* EntryPointPolymorphicInlineCacheInfo::EnsureInlineeInfo(Recycler \* recycler, FunctionBody \* inlineeFunctionBody)

{

PolymorphicInlineCacheInfo \* info = GetInlineeInfo(inlineeFunctionBody);

if (!info)

{

info = RecyclerNew(recycler, PolymorphicInlineCacheInfo, inlineeFunctionBody);

inlineeInfo.Prepend(recycler, info);

}

return info;

}

void EntryPointPolymorphicInlineCacheInfo::SetPolymorphicInlineCache(FunctionBody \* functionBody, uint index, PolymorphicInlineCache \* polymorphicInlineCache, bool isInlinee, byte polyCacheUtil)

{

if (!isInlinee)

{

SetPolymorphicInlineCache(&selfInfo, functionBody, index, polymorphicInlineCache, polyCacheUtil);

Assert(functionBody == selfInfo.GetFunctionBody());

}

else

{

SetPolymorphicInlineCache(EnsureInlineeInfo(functionBody->GetScriptContext()->GetRecycler(), functionBody), functionBody, index, polymorphicInlineCache, polyCacheUtil);

Assert(functionBody == GetInlineeInfo(functionBody)->GetFunctionBody());

}

}

void EntryPointPolymorphicInlineCacheInfo::SetPolymorphicInlineCache(PolymorphicInlineCacheInfo \* polymorphicInlineCacheInfo, FunctionBody \* functionBody, uint index, PolymorphicInlineCache \* polymorphicInlineCache, byte polyCacheUtil)

{

polymorphicInlineCacheInfo->GetPolymorphicInlineCaches()->SetInlineCache(functionBody->GetScriptContext()->GetRecycler(), functionBody, index, polymorphicInlineCache);

polymorphicInlineCacheInfo->GetUtilArray()->SetUtil(functionBody, index, polyCacheUtil);

}

void PolymorphicCacheUtilizationArray::SetUtil(Js::FunctionBody\* functionBody, uint index, byte util)

{

Assert(functionBody);

Assert(index < functionBody->GetInlineCacheCount());

EnsureUtilArray(functionBody->GetScriptContext()->GetRecycler(), functionBody);

this->utilArray[index] = util;

}

byte PolymorphicCacheUtilizationArray::GetUtil(Js::FunctionBody\* functionBody, uint index)

{

Assert(index < functionBody->GetInlineCacheCount());

return this->utilArray[index];

}

void PolymorphicCacheUtilizationArray::EnsureUtilArray(Recycler \* const recycler, Js::FunctionBody \* functionBody)

{

Assert(recycler);

Assert(functionBody);

Assert(functionBody->GetInlineCacheCount() != 0);

if(this->utilArray)

{

return;

}

this->utilArray = RecyclerNewArrayZ(recycler, byte, functionBody->GetInlineCacheCount());

}

#if ENABLE\_NATIVE\_CODEGEN

void EntryPointInfo::AddWeakFuncRef(RecyclerWeakReference<FunctionBody> \*weakFuncRef, Recycler \*recycler)

{

Assert(this->state == CodeGenPending);

this->weakFuncRefSet = this->EnsureWeakFuncRefSet(recycler);

this->weakFuncRefSet->AddNew(weakFuncRef);

}

EntryPointInfo::WeakFuncRefSet \*

EntryPointInfo::EnsureWeakFuncRefSet(Recycler \*recycler)

{

if (this->weakFuncRefSet == nullptr)

{

this->weakFuncRefSet = RecyclerNew(recycler, WeakFuncRefSet, recycler);

}

return this->weakFuncRefSet;

}

void EntryPointInfo::EnsureIsReadyToCall()

{

ProcessJitTransferData();

}

void EntryPointInfo::ProcessJitTransferData()

{

Assert(!IsCleanedUp());

if (GetJitTransferData() != nullptr && GetJitTransferData()->GetIsReady())

{

class AutoCleanup

{

EntryPointInfo \*entryPointInfo;

public:

AutoCleanup(EntryPointInfo \*entryPointInfo) : entryPointInfo(entryPointInfo)

{

}

void Done()

{

entryPointInfo = nullptr;

}

~AutoCleanup()

{

if (entryPointInfo)

{

entryPointInfo->OnNativeCodeInstallFailure();

}

}

} autoCleanup(this);

ScriptContext\* scriptContext = GetScriptContext();

PinTypeRefs(scriptContext);

InstallGuards(scriptContext);

FreeJitTransferData();

autoCleanup.Done();

}

}

EntryPointInfo::JitTransferData\* EntryPointInfo::EnsureJitTransferData(Recycler\* recycler)

{

if (this->jitTransferData == nullptr)

{

this->jitTransferData = RecyclerNew(recycler, EntryPointInfo::JitTransferData);

}

return this->jitTransferData;

}

#ifdef FIELD\_ACCESS\_STATS

FieldAccessStats\* EntryPointInfo::EnsureFieldAccessStats(Recycler\* recycler)

{

if (this->fieldAccessStats == nullptr)

{

this->fieldAccessStats = RecyclerNew(recycler, FieldAccessStats);

}

return this->fieldAccessStats;

}

#endif

void EntryPointInfo::JitTransferData::AddJitTimeTypeRef(void\* typeRef, Recycler\* recycler)

{

Assert(typeRef != nullptr);

EnsureJitTimeTypeRefs(recycler);

this->jitTimeTypeRefs->AddNew(typeRef);

}

void EntryPointInfo::JitTransferData::EnsureJitTimeTypeRefs(Recycler\* recycler)

{

if (this->jitTimeTypeRefs == nullptr)

{

this->jitTimeTypeRefs = RecyclerNew(recycler, TypeRefSet, recycler);

}

}

void EntryPointInfo::PinTypeRefs(ScriptContext\* scriptContext)

{

Assert(this->jitTransferData != nullptr && this->jitTransferData->GetIsReady());

Recycler\* recycler = scriptContext->GetRecycler();

if (this->jitTransferData->GetRuntimeTypeRefs() != nullptr)

{

// Copy pinned types from a heap allocated array created on the background thread

// to a recycler allocated array which will live as long as this EntryPointInfo.

// The original heap allocated array will be freed at the end of NativeCodeGenerator::CheckCodeGenDone

void\*\* jitPinnedTypeRefs = this->jitTransferData->GetRuntimeTypeRefs();

size\_t jitPinnedTypeRefCount = this->jitTransferData->GetRuntimeTypeRefCount();

this->runtimeTypeRefs = RecyclerNewArray(recycler, void\*, jitPinnedTypeRefCount + 1);

js\_memcpy\_s(this->runtimeTypeRefs, jitPinnedTypeRefCount \* sizeof(void\*), jitPinnedTypeRefs, jitPinnedTypeRefCount \* sizeof(void\*));

this->runtimeTypeRefs[jitPinnedTypeRefCount] = nullptr;

}

}

void EntryPointInfo::InstallGuards(ScriptContext\* scriptContext)

{

Assert(this->jitTransferData != nullptr && this->jitTransferData->GetIsReady());

Assert(this->equivalentTypeCacheCount == 0 && this->equivalentTypeCaches == nullptr);

Assert(this->propertyGuardCount == 0 && this->propertyGuardWeakRefs == nullptr);

class AutoCleanup

{

EntryPointInfo \*entryPointInfo;

public:

AutoCleanup(EntryPointInfo \*entryPointInfo) : entryPointInfo(entryPointInfo)

{

}

void Done()

{

entryPointInfo = nullptr;

}

~AutoCleanup()

{

if (entryPointInfo)

{

entryPointInfo->equivalentTypeCacheCount = 0;

entryPointInfo->equivalentTypeCaches = nullptr;

entryPointInfo->propertyGuardCount = 0;

entryPointInfo->propertyGuardWeakRefs = nullptr;

entryPointInfo->UnregisterEquivalentTypeCaches();

}

}

} autoCleanup(this);

for (int i = 0; i < this->jitTransferData->lazyBailoutPropertyCount; i++)

{

Assert(this->jitTransferData->lazyBailoutProperties != nullptr);

Js::PropertyId propertyId = this->jitTransferData->lazyBailoutProperties[i];

Js::PropertyGuard\* sharedPropertyGuard;

bool hasSharedPropertyGuard = TryGetSharedPropertyGuard(propertyId, sharedPropertyGuard);

Assert(hasSharedPropertyGuard);

bool isValid = hasSharedPropertyGuard ? sharedPropertyGuard->IsValid() : false;

if (isValid)

{

scriptContext->GetThreadContext()->RegisterLazyBailout(propertyId, this);

}

else

{

OUTPUT\_TRACE2(Js::LazyBailoutPhase, this->GetFunctionBody(), L"Lazy bailout - Invalidation due to property: %s \n", scriptContext->GetPropertyName(propertyId)->GetBuffer());

this->Invalidate(true);

return;

}

}

if (this->jitTransferData->equivalentTypeGuardCount > 0)

{

Assert(this->jitTransferData->equivalentTypeGuards != nullptr);

Recycler\* recycler = scriptContext->GetRecycler();

int guardCount = this->jitTransferData->equivalentTypeGuardCount;

JitEquivalentTypeGuard\*\* guards = this->jitTransferData->equivalentTypeGuards;

// Create an array of equivalent type caches on the entry point info to ensure they are kept

// alive for the lifetime of the entry point.

this->equivalentTypeCacheCount = guardCount;

// No need to zero-initialize, since we will populate all data slots.

// We used to let the recycler scan the types in the cache, but we no longer do. See

// ThreadContext::ClearEquivalentTypeCaches for an explanation.

this->equivalentTypeCaches = RecyclerNewArrayLeafZ(recycler, EquivalentTypeCache, guardCount);

this->RegisterEquivalentTypeCaches();

EquivalentTypeCache\* cache = this->equivalentTypeCaches;

for (JitEquivalentTypeGuard\*\* guard = guards; guard < guards + guardCount; guard++)

{

EquivalentTypeCache\* oldCache = (\*guard)->GetCache();

// Copy the contents of the heap-allocated cache to the recycler-allocated version to make sure the types are

// kept alive. Allow the properties pointer to refer to the heap-allocated arrays. It will stay alive as long

// as the entry point is alive, and property entries contain no pointers to other recycler allocated objects.

(\*cache) = (\*oldCache);

// Set the recycler-allocated cache on the (heap-allocated) guard.

(\*guard)->SetCache(cache);

cache++;

}

}

// The propertyGuardsByPropertyId structure is temporary and serves only to register the type guards for the correct

// properties. If we've done code gen for this EntryPointInfo, typePropertyGuardsByPropertyId will have been used and nulled out.

if (this->jitTransferData->propertyGuardsByPropertyId != nullptr)

{

this->propertyGuardCount = this->jitTransferData->propertyGuardCount;

this->propertyGuardWeakRefs = RecyclerNewArrayZ(scriptContext->GetRecycler(), FakePropertyGuardWeakReference\*, this->propertyGuardCount);

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

Js::TypeGuardTransferEntry\* entry = this->jitTransferData->propertyGuardsByPropertyId;

while (entry->propertyId != Js::Constants::NoProperty)

{

Js::PropertyId propertyId = entry->propertyId;

Js::PropertyGuard\* sharedPropertyGuard;

// We use the shared guard created during work item creation to ensure that the condition we assumed didn't change while

// we were JIT-ing. If we don't have a shared property guard for this property then we must not need to protect it,

// because it exists on the instance. Unfortunately, this means that if we have a bug and fail to create a shared

// guard for some property during work item creation, we won't find out about it here.

bool isNeeded = TryGetSharedPropertyGuard(propertyId, sharedPropertyGuard);

bool isValid = isNeeded ? sharedPropertyGuard->IsValid() : false;

int entryGuardIndex = 0;

while (entry->guards[entryGuardIndex] != nullptr)

{

if (isNeeded)

{

Js::JitIndexedPropertyGuard\* guard = entry->guards[entryGuardIndex];

int guardIndex = guard->GetIndex();

Assert(guardIndex >= 0 && guardIndex < this->propertyGuardCount);

// We use the shared guard here to make sure the conditions we assumed didn't change while we were JIT-ing.

// If they did, we proactively invalidate the guard here, so that we bail out if we try to call this code.

if (isValid)

{

auto propertyGuardWeakRef = this->propertyGuardWeakRefs[guardIndex];

if (propertyGuardWeakRef == nullptr)

{

propertyGuardWeakRef = Js::FakePropertyGuardWeakReference::New(scriptContext->GetRecycler(), guard);

this->propertyGuardWeakRefs[guardIndex] = propertyGuardWeakRef;

}

Assert(propertyGuardWeakRef->Get() == guard);

threadContext->RegisterUniquePropertyGuard(propertyId, propertyGuardWeakRef);

}

else

{

guard->Invalidate();

}

}

entryGuardIndex++;

}

entry = reinterpret\_cast<Js::TypeGuardTransferEntry\*>(&entry->guards[++entryGuardIndex]);

}

}

// The ctorCacheGuardsByPropertyId structure is temporary and serves only to register the constructor cache guards for the correct

// properties. If we've done code gen for this EntryPointInfo, ctorCacheGuardsByPropertyId will have been used and nulled out.

// Unlike type property guards, constructor cache guards use the live constructor caches associated with function objects. These are

// recycler allocated and are kept alive by the constructorCaches field, where they were inserted during work item creation.

if (this->jitTransferData->ctorCacheGuardsByPropertyId != nullptr)

{

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

Js::CtorCacheGuardTransferEntry\* entry = this->jitTransferData->ctorCacheGuardsByPropertyId;

while (entry->propertyId != Js::Constants::NoProperty)

{

Js::PropertyId propertyId = entry->propertyId;

Js::PropertyGuard\* sharedPropertyGuard;

// We use the shared guard created during work item creation to ensure that the condition we assumed didn't change while

// we were JIT-ing. If we don't have a shared property guard for this property then we must not need to protect it,

// because it exists on the instance. Unfortunately, this means that if we have a bug and fail to create a shared

// guard for some property during work item creation, we won't find out about it here.

bool isNeeded = TryGetSharedPropertyGuard(propertyId, sharedPropertyGuard);

bool isValid = isNeeded ? sharedPropertyGuard->IsValid() : false;

int entryCacheIndex = 0;

while (entry->caches[entryCacheIndex] != nullptr)

{

if (isNeeded)

{

Js::ConstructorCache\* cache = entry->caches[entryCacheIndex];

// We use the shared cache here to make sure the conditions we assumed didn't change while we were JIT-ing.

// If they did, we proactively invalidate the cache here, so that we bail out if we try to call this code.

if (isValid)

{

threadContext->RegisterConstructorCache(propertyId, cache);

}

else

{

cache->InvalidateAsGuard();

}

}

entryCacheIndex++;

}

entry = reinterpret\_cast<Js::CtorCacheGuardTransferEntry\*>(&entry->caches[++entryCacheIndex]);

}

}

if (PHASE\_ON(Js::FailNativeCodeInstallPhase, this->GetFunctionBody()))

{

Js::Throw::OutOfMemory();

}

autoCleanup.Done();

}

PropertyGuard\* EntryPointInfo::RegisterSharedPropertyGuard(Js::PropertyId propertyId, ScriptContext\* scriptContext)

{

if (this->sharedPropertyGuards == nullptr)

{

Recycler\* recycler = scriptContext->GetRecycler();

this->sharedPropertyGuards = RecyclerNew(recycler, SharedPropertyGuardDictionary, recycler);

}

PropertyGuard\* guard;

if (!this->sharedPropertyGuards->TryGetValue(propertyId, &guard))

{

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

guard = threadContext->RegisterSharedPropertyGuard(propertyId);

this->sharedPropertyGuards->Add(propertyId, guard);

}

return guard;

}

bool EntryPointInfo::HasSharedPropertyGuard(Js::PropertyId propertyId)

{

return this->sharedPropertyGuards != nullptr ? this->sharedPropertyGuards->ContainsKey(propertyId) : false;

}

bool EntryPointInfo::TryGetSharedPropertyGuard(Js::PropertyId propertyId, Js::PropertyGuard\*& guard)

{

return this->sharedPropertyGuards != nullptr ? this->sharedPropertyGuards->TryGetValue(propertyId, &guard) : false;

}

void EntryPointInfo::RecordTypeGuards(int typeGuardCount, TypeGuardTransferEntry\* typeGuardTransferRecord, size\_t typeGuardTransferPlusSize)

{

Assert(this->jitTransferData != nullptr);

this->jitTransferData->propertyGuardCount = typeGuardCount;

this->jitTransferData->propertyGuardsByPropertyId = typeGuardTransferRecord;

this->jitTransferData->propertyGuardsByPropertyIdPlusSize = typeGuardTransferPlusSize;

}

void EntryPointInfo::RecordCtorCacheGuards(CtorCacheGuardTransferEntry\* ctorCacheTransferRecord, size\_t ctorCacheTransferPlusSize)

{

Assert(this->jitTransferData != nullptr);

this->jitTransferData->ctorCacheGuardsByPropertyId = ctorCacheTransferRecord;

this->jitTransferData->ctorCacheGuardsByPropertyIdPlusSize = ctorCacheTransferPlusSize;

}

void EntryPointInfo::FreePropertyGuards()

{

// While typePropertyGuardWeakRefs are allocated via NativeCodeData::Allocator and will be automatically freed to the heap,

// we must zero out the fake weak references so that property guard invalidation doesn't access freed memory.

if (this->propertyGuardWeakRefs != nullptr)

{

for (int i = 0; i < this->propertyGuardCount; i++)

{

if (this->propertyGuardWeakRefs[i] != nullptr)

{

this->propertyGuardWeakRefs[i]->Zero();

}

}

this->propertyGuardCount = 0;

this->propertyGuardWeakRefs = nullptr;

}

}

void EntryPointInfo::RecordBailOutMap(JsUtil::List<LazyBailOutRecord, ArenaAllocator>\* bailoutMap)

{

Assert(this->bailoutRecordMap == nullptr);

this->bailoutRecordMap = HeapNew(BailOutRecordMap, &HeapAllocator::Instance);

this->bailoutRecordMap->Copy(bailoutMap);

}

void EntryPointInfo::RecordInlineeFrameMap(JsUtil::List<NativeOffsetInlineeFramePair, ArenaAllocator>\* tempInlineeFrameMap)

{

Assert(this->inlineeFrameMap == nullptr);

if (tempInlineeFrameMap->Count() > 0)

{

this->inlineeFrameMap = HeapNew(InlineeFrameMap, &HeapAllocator::Instance);

this->inlineeFrameMap->Copy(tempInlineeFrameMap);

}

}

InlineeFrameRecord\* EntryPointInfo::FindInlineeFrame(void\* returnAddress)

{

if (this->inlineeFrameMap == nullptr)

{

return nullptr;

}

size\_t offset = (size\_t)((BYTE\*)returnAddress - (BYTE\*)this->GetNativeAddress());

int index = this->inlineeFrameMap->BinarySearch([=](const NativeOffsetInlineeFramePair& pair, int index) {

if (pair.offset >= offset)

{

if (index == 0 || index > 0 && this->inlineeFrameMap->Item(index - 1).offset < offset)

{

return 0;

}

else

{

return 1;

}

}

return -1;

});

if (index == -1)

{

return nullptr;

}

return this->inlineeFrameMap->Item(index).record;

}

void EntryPointInfo::DoLazyBailout(BYTE\*\* addressOfInstructionPointer, Js::FunctionBody\* functionBody, const PropertyRecord\* propertyRecord)

{

BYTE\* instructionPointer = \*addressOfInstructionPointer;

Assert(instructionPointer > (BYTE\*)this->nativeAddress && instructionPointer < ((BYTE\*)this->nativeAddress + this->codeSize));

size\_t offset = instructionPointer - (BYTE\*)this->nativeAddress;

LazyBailOutRecord record;

int found = this->bailoutRecordMap->BinarySearch([=](const LazyBailOutRecord& record, int index)

{

// find the closest entry which is greater than the current offset.

if (record.offset >= offset)

{

if (index == 0 || index > 0 && this->bailoutRecordMap->Item(index - 1).offset < offset)

{

return 0;

}

else

{

return 1;

}

}

return -1;

});

if (found != -1)

{

LazyBailOutRecord& record = this->bailoutRecordMap->Item(found);

\*addressOfInstructionPointer = record.instructionPointer;

record.SetBailOutKind();

if (PHASE\_TRACE1(Js::LazyBailoutPhase))

{

Output::Print(L"On stack lazy bailout. Property: %s Old IP: 0x%x New IP: 0x%x ", propertyRecord->GetBuffer(), instructionPointer, record.instructionPointer);

#if DBG

record.Dump(functionBody);

#endif

Output::Print(L"\n");

}

}

else

{

AssertMsg(false, "Lazy Bailout address mapping missing");

}

}

void EntryPointInfo::FreeJitTransferData()

{

JitTransferData\* jitTransferData = this->jitTransferData;

this->jitTransferData = nullptr;

if (jitTransferData != nullptr)

{

// This dictionary is recycler allocated so it doesn't need to be explicitly freed.

jitTransferData->jitTimeTypeRefs = nullptr;

if (jitTransferData->lazyBailoutProperties != nullptr)

{

HeapDeleteArray(jitTransferData->lazyBailoutPropertyCount, jitTransferData->lazyBailoutProperties);

jitTransferData->lazyBailoutProperties = nullptr;

}

// All structures below are heap allocated and need to be freed explicitly.

if (jitTransferData->runtimeTypeRefs != nullptr)

{

HeapDeleteArray(jitTransferData->runtimeTypeRefCount, jitTransferData->runtimeTypeRefs);

jitTransferData->runtimeTypeRefs = nullptr;

}

jitTransferData->runtimeTypeRefCount = 0;

if (jitTransferData->propertyGuardsByPropertyId != nullptr)

{

HeapDeletePlus(jitTransferData->propertyGuardsByPropertyIdPlusSize, jitTransferData->propertyGuardsByPropertyId);

jitTransferData->propertyGuardsByPropertyId = nullptr;

}

jitTransferData->propertyGuardCount = 0;

jitTransferData->propertyGuardsByPropertyIdPlusSize = 0;

if (jitTransferData->ctorCacheGuardsByPropertyId != nullptr)

{

HeapDeletePlus(jitTransferData->ctorCacheGuardsByPropertyIdPlusSize, jitTransferData->ctorCacheGuardsByPropertyId);

jitTransferData->ctorCacheGuardsByPropertyId = nullptr;

}

jitTransferData->ctorCacheGuardsByPropertyIdPlusSize = 0;

if (jitTransferData->equivalentTypeGuards != nullptr)

{

HeapDeleteArray(jitTransferData->equivalentTypeGuardCount, jitTransferData->equivalentTypeGuards);

jitTransferData->equivalentTypeGuards = nullptr;

}

jitTransferData->equivalentTypeGuardCount = 0;

if (jitTransferData->data != nullptr)

{

HeapDelete(jitTransferData->data);

jitTransferData->data = nullptr;

}

jitTransferData = nullptr;

}

}

void EntryPointInfo::RegisterEquivalentTypeCaches()

{

Assert(this->registeredEquivalentTypeCacheRef == nullptr);

this->registeredEquivalentTypeCacheRef =

GetScriptContext()->GetThreadContext()->RegisterEquivalentTypeCacheEntryPoint(this);

}

void EntryPointInfo::UnregisterEquivalentTypeCaches()

{

if (this->registeredEquivalentTypeCacheRef != nullptr)

{

ScriptContext \*scriptContext = GetScriptContext();

if (scriptContext != nullptr)

{

scriptContext->GetThreadContext()->UnregisterEquivalentTypeCacheEntryPoint(

this->registeredEquivalentTypeCacheRef);

}

this->registeredEquivalentTypeCacheRef = nullptr;

}

}

bool EntryPointInfo::ClearEquivalentTypeCaches()

{

Assert(this->equivalentTypeCaches != nullptr);

Assert(this->equivalentTypeCacheCount > 0);

bool isAnyCacheLive = false;

Recycler \*recycler = GetScriptContext()->GetRecycler();

for (EquivalentTypeCache \*cache = this->equivalentTypeCaches;

cache < this->equivalentTypeCaches + this->equivalentTypeCacheCount;

cache++)

{

bool isCacheLive = cache->ClearUnusedTypes(recycler);

if (isCacheLive)

{

isAnyCacheLive = true;

}

}

if (!isAnyCacheLive)

{

// The caller must take care of unregistering this entry point. We may be in the middle of

// walking the list of registered entry points.

this->equivalentTypeCaches = nullptr;

this->equivalentTypeCacheCount = 0;

this->registeredEquivalentTypeCacheRef = nullptr;

}

return isAnyCacheLive;

}

bool EquivalentTypeCache::ClearUnusedTypes(Recycler \*recycler)

{

bool isAnyTypeLive = false;

Assert(this->guard);

if (this->guard->IsValid())

{

Type \*type = reinterpret\_cast<Type\*>(this->guard->GetValue());

if (!recycler->IsObjectMarked(type))

{

this->guard->Invalidate();

}

else

{

isAnyTypeLive = true;

}

}

for (int i = 0; i < EQUIVALENT\_TYPE\_CACHE\_SIZE; i++)

{

Type \*type = this->types[i];

if (type != nullptr)

{

if (!recycler->IsObjectMarked(type))

{

this->types[i] = nullptr;

}

else

{

isAnyTypeLive = true;

}

}

}

return isAnyTypeLive;

}

void EntryPointInfo::RegisterConstructorCache(Js::ConstructorCache\* constructorCache, Recycler\* recycler)

{

Assert(constructorCache != nullptr);

if (!this->constructorCaches)

{

this->constructorCaches = RecyclerNew(recycler, ConstructorCacheList, recycler);

}

this->constructorCaches->Prepend(constructorCache);

}

#endif

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

void EntryPointInfo::CaptureCleanupStackTrace()

{

if (this->cleanupStack != nullptr)

{

this->cleanupStack->Delete(&NoCheckHeapAllocator::Instance);

this->cleanupStack = nullptr;

}

this->cleanupStack = StackBackTrace::Capture(&NoCheckHeapAllocator::Instance);

}

#endif

void EntryPointInfo::Finalize(bool isShutdown)

{

\_\_super::Finalize(isShutdown);

if (!isShutdown)

{

ReleasePendingWorkItem();

}

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

this->SetCleanupReason(CleanupReason::CleanUpForFinalize);

#endif

this->Cleanup(isShutdown, false);

#if DBG

if (this->cleanupStack != nullptr)

{

this->cleanupStack->Delete(&NoCheckHeapAllocator::Instance);

this->cleanupStack = nullptr;

}

#endif

this->library = nullptr;

}

#if ENABLE\_NATIVE\_CODEGEN

EntryPointPolymorphicInlineCacheInfo \* EntryPointInfo::EnsurePolymorphicInlineCacheInfo(Recycler \* recycler, FunctionBody \* functionBody)

{

if (!polymorphicInlineCacheInfo)

{

polymorphicInlineCacheInfo = RecyclerNew(recycler, EntryPointPolymorphicInlineCacheInfo, functionBody);

}

return polymorphicInlineCacheInfo;

}

#endif

void EntryPointInfo::Cleanup(bool isShutdown, bool captureCleanupStack)

{

if (this->GetState() != CleanedUp)

{

this->OnCleanup(isShutdown);

#if ENABLE\_NATIVE\_CODEGEN

FreeJitTransferData();

if (this->bailoutRecordMap != nullptr)

{

HeapDelete(this->bailoutRecordMap);

bailoutRecordMap = nullptr;

}

if (this->sharedPropertyGuards != nullptr)

{

sharedPropertyGuards->Clear();

sharedPropertyGuards = nullptr;

}

FreePropertyGuards();

if (this->equivalentTypeCaches != nullptr)

{

this->UnregisterEquivalentTypeCaches();

this->equivalentTypeCacheCount = 0;

this->equivalentTypeCaches = nullptr;

}

if (this->constructorCaches != nullptr)

{

this->constructorCaches->Clear();

}

#endif

// This is how we set the CleanedUp state

this->workItem = nullptr;

this->nativeAddress = nullptr;

#if ENABLE\_NATIVE\_CODEGEN

this->weakFuncRefSet = nullptr;

this->runtimeTypeRefs = nullptr;

#endif

this->codeSize = -1;

this->library = nullptr;

#if ENABLE\_NATIVE\_CODEGEN

DeleteNativeCodeData(this->data);

this->data = nullptr;

this->numberChunks = nullptr;

#endif

this->state = CleanedUp;

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

#if !DBG

captureCleanupStack = captureCleanupStack && Js::Configuration::Global.flags.FreTestDiagMode;

#endif

if (captureCleanupStack)

{

this->CaptureCleanupStackTrace();

}

#endif

#if ENABLE\_NATIVE\_CODEGEN

if (nullptr != this->nativeThrowSpanSequence)

{

HeapDelete(this->nativeThrowSpanSequence);

this->nativeThrowSpanSequence = nullptr;

}

this->polymorphicInlineCacheInfo = nullptr;

#endif

#if DBG\_DUMP | defined(VTUNE\_PROFILING)

this->nativeOffsetMaps.Reset();

#endif

}

}

void EntryPointInfo::Reset(bool resetStateToNotScheduled)

{

Assert(this->GetState() != CleanedUp);

this->nativeAddress = nullptr;

this->workItem = nullptr;

#if ENABLE\_NATIVE\_CODEGEN

if (nullptr != this->nativeThrowSpanSequence)

{

HeapDelete(this->nativeThrowSpanSequence);

this->nativeThrowSpanSequence = nullptr;

}

#endif

this->codeSize = 0;

#if ENABLE\_NATIVE\_CODEGEN

this->weakFuncRefSet = nullptr;

this->sharedPropertyGuards = nullptr;

FreePropertyGuards();

FreeJitTransferData();

if (this->data != nullptr)

{

DeleteNativeCodeData(this->data);

this->data = nullptr;

}

#endif

// Set the state to NotScheduled only if the call to Reset is not because of JIT cap being reached

if (resetStateToNotScheduled)

{

this->state = NotScheduled;

}

}

#if ENABLE\_NATIVE\_CODEGEN

void EntryPointInfo::ResetOnNativeCodeInstallFailure()

{

// Reset the entry point without attempting to create a new default and GenerateFunction on it.

// Do this for LoopEntryPointInfo or if we throw during FunctionEntryPointInfo::Invalidate.

this->Reset(true);

Assert(this->address != nullptr);

FreeNativeCodeGenAllocation(GetScriptContext(), this->address);

this->address = nullptr;

}

#endif

#ifdef PERF\_COUNTERS

void FunctionEntryPointInfo::OnRecorded()

{

PERF\_COUNTER\_ADD(Code, TotalNativeCodeSize, GetCodeSize());

PERF\_COUNTER\_ADD(Code, FunctionNativeCodeSize, GetCodeSize());

PERF\_COUNTER\_ADD(Code, DynamicNativeCodeSize, GetCodeSize());

}

#endif

FunctionEntryPointInfo::FunctionEntryPointInfo(FunctionProxy \* functionProxy, void \* address, ThreadContext\* context, void\* cookie) :

EntryPointInfo(address, functionProxy->GetScriptContext()->GetLibrary(), cookie, context),

localVarSlotsOffset(Js::Constants::InvalidOffset),

localVarChangedOffset(Js::Constants::InvalidOffset),

callsCount(0),

jitMode(ExecutionMode::Interpreter),

nativeEntryPointProcessed(false),

functionProxy(functionProxy),

nextEntryPoint(nullptr),

mIsTemplatizedJitMode(false)

{

}

#ifndef TEMP\_DISABLE\_ASMJS

void FunctionEntryPointInfo::SetOldFunctionEntryPointInfo(FunctionEntryPointInfo\* entrypointInfo)

{

Assert(this->GetIsAsmJSFunction());

Assert(entrypointInfo);

mOldFunctionEntryPointInfo = entrypointInfo;

};

FunctionEntryPointInfo\* FunctionEntryPointInfo::GetOldFunctionEntryPointInfo()const

{

Assert(this->GetIsAsmJSFunction());

return mOldFunctionEntryPointInfo;

};

void FunctionEntryPointInfo::SetIsTJMode(bool value)

{

Assert(this->GetIsAsmJSFunction());

mIsTemplatizedJitMode = value;

}

bool FunctionEntryPointInfo::GetIsTJMode()const

{

return mIsTemplatizedJitMode;

};

#endif

//End AsmJS Support

#if ENABLE\_NATIVE\_CODEGEN

ExecutionMode FunctionEntryPointInfo::GetJitMode() const

{

return jitMode;

}

void FunctionEntryPointInfo::SetJitMode(const ExecutionMode jitMode)

{

Assert(jitMode == ExecutionMode::SimpleJit || jitMode == ExecutionMode::FullJit);

this->jitMode = jitMode;

}

#endif

void FunctionEntryPointInfo::ReleasePendingWorkItem()

{

// Do this outside of Cleanup since cleanup can be called from the background thread

// We remove any work items corresponding to the function body being reclaimed

// so that the background thread doesn't try to use them. ScriptContext != null => this

// is a function entry point

// In general this is not needed for loop bodies since loop bodies aren't in the low priority

// queue, they should be jitted before the entry point is finalized

if (!this->IsNotScheduled() && !this->IsCleanedUp())

{

#if defined(\_M\_ARM32\_OR\_ARM64)

// On ARM machines, order of writes is not guaranteed while reading data from another processor

// So we need to have a memory barrier here in order to make sure that the work item is consistent

MemoryBarrier();

#endif

CodeGenWorkItem\* workItem = this->GetWorkItem();

if (workItem != nullptr)

{

Assert(this->library != nullptr);

#if ENABLE\_NATIVE\_CODEGEN

TryReleaseNonHiPriWorkItem(this->library->GetScriptContext(), workItem);

#endif

}

}

}

FunctionBody \*FunctionEntryPointInfo::GetFunctionBody() const

{

return functionProxy->GetFunctionBody();

}

void FunctionEntryPointInfo::OnCleanup(bool isShutdown)

{

if (this->IsCodeGenDone())

{

Assert(this->functionProxy->HasBody());

#if ENABLE\_NATIVE\_CODEGEN

if (nullptr != this->inlineeFrameMap)

{

HeapDelete(this->inlineeFrameMap);

this->inlineeFrameMap = nullptr;

}

#endif

if(nativeEntryPointProcessed)

{

JS\_ETW(EtwTrace::LogMethodNativeUnloadEvent(this->functionProxy->GetFunctionBody(), this));

}

FunctionBody\* functionBody = this->functionProxy->GetFunctionBody();

#ifndef TEMP\_DISABLE\_ASMJS

if (this->GetIsTJMode())

{

// release LoopHeaders here if the entrypointInfo is TJ

this->GetFunctionBody()->ReleaseLoopHeaders();

}

#endif

if(functionBody->GetSimpleJitEntryPointInfo() == this)

{

functionBody->SetSimpleJitEntryPointInfo(nullptr);

}

// If we're shutting down, the script context might be gone

if (!isShutdown)

{

ScriptContext\* scriptContext = this->functionProxy->GetScriptContext();

void\* currentCookie = nullptr;

#if ENABLE\_NATIVE\_CODEGEN

// In the debugger case, we might call cleanup after the native code gen that

// allocated this entry point has already shutdown. In that case, the validation

// check below should fail and we should not try to free this entry point

// since it's already been freed

NativeCodeGenerator\* currentNativeCodegen = scriptContext->GetNativeCodeGenerator();

Assert(this->validationCookie != nullptr);

currentCookie = (void\*)currentNativeCodegen;

#endif

if (validationCookie == currentCookie)

{

scriptContext->FreeFunctionEntryPoint((Js::JavascriptMethod)this->GetNativeAddress());

}

}

#ifdef PERF\_COUNTERS

PERF\_COUNTER\_SUB(Code, TotalNativeCodeSize, GetCodeSize());

PERF\_COUNTER\_SUB(Code, FunctionNativeCodeSize, GetCodeSize());

PERF\_COUNTER\_SUB(Code, DynamicNativeCodeSize, GetCodeSize());

#endif

}

this->functionProxy = nullptr;

}

#if ENABLE\_NATIVE\_CODEGEN

void FunctionEntryPointInfo::OnNativeCodeInstallFailure()

{

this->Invalidate(false);

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

this->SetCleanupReason(CleanupReason::NativeCodeInstallFailure);

#endif

this->Cleanup(false, true /\* capture cleanup stack \*/);

}

void FunctionEntryPointInfo::EnterExpirableCollectMode()

{

this->lastCallsCount = this->callsCount;

// For code that is not jitted yet we don't want to expire since there is nothing to free here

if (this->IsCodeGenPending())

{

this->SetIsObjectUsed();

}

}

void FunctionEntryPointInfo::Invalidate(bool prolongEntryPoint)

{

Assert(!this->functionProxy->IsDeferred());

FunctionBody\* functionBody = this->functionProxy->GetFunctionBody();

Assert(this != functionBody->GetSimpleJitEntryPointInfo());

// We may have got here following OOM in ProcessJitTransferData. Free any data we have

// to reduce the chance of another OOM below.

this->FreeJitTransferData();

FunctionEntryPointInfo\* entryPoint = functionBody->GetDefaultFunctionEntryPointInfo();

if (entryPoint->IsCodeGenPending())

{

OUTPUT\_TRACE(Js::LazyBailoutPhase, L"Skipping creating new entrypoint as one is already pending\n");

}

else

{

class AutoCleanup

{

EntryPointInfo \*entryPointInfo;

public:

AutoCleanup(EntryPointInfo \*entryPointInfo) : entryPointInfo(entryPointInfo)

{

}

void Done()

{

entryPointInfo = nullptr;

}

~AutoCleanup()

{

if (entryPointInfo)

{

entryPointInfo->ResetOnNativeCodeInstallFailure();

}

}

} autoCleanup(this);

entryPoint = functionBody->CreateNewDefaultEntryPoint();

GenerateFunction(functionBody->GetScriptContext()->GetNativeCodeGenerator(), functionBody, /\*function\*/ nullptr);

autoCleanup.Done();

}

this->functionProxy->MapFunctionObjectTypes([&](DynamicType\* type)

{

Assert(type->GetTypeId() == TypeIds\_Function);

ScriptFunctionType\* functionType = (ScriptFunctionType\*)type;

if (functionType->GetEntryPointInfo() == this)

{

functionType->SetEntryPointInfo(entryPoint);

functionType->SetEntryPoint(this->functionProxy->GetDirectEntryPoint(entryPoint));

}

});

if (!prolongEntryPoint)

{

ThreadContext\* threadContext = this->functionProxy->GetScriptContext()->GetThreadContext();

threadContext->QueueFreeOldEntryPointInfoIfInScript(this);

}

}

void FunctionEntryPointInfo::Expire()

{

if (this->lastCallsCount != this->callsCount || !this->nativeEntryPointProcessed || this->IsCleanedUp())

{

return;

}

ThreadContext\* threadContext = this->functionProxy->GetScriptContext()->GetThreadContext();

Assert(!this->functionProxy->IsDeferred());

FunctionBody\* functionBody = this->functionProxy->GetFunctionBody();

FunctionEntryPointInfo \*simpleJitEntryPointInfo = functionBody->GetSimpleJitEntryPointInfo();

const bool expiringSimpleJitEntryPointInfo = simpleJitEntryPointInfo == this;

if(expiringSimpleJitEntryPointInfo)

{

if(functionBody->GetExecutionMode() != ExecutionMode::FullJit)

{

// Don't expire simple JIT code until the transition to full JIT

return;

}

simpleJitEntryPointInfo = nullptr;

functionBody->SetSimpleJitEntryPointInfo(nullptr);

}

try

{

AUTO\_NESTED\_HANDLED\_EXCEPTION\_TYPE(ExceptionType\_OutOfMemory);

FunctionEntryPointInfo\* newEntryPoint = nullptr;

FunctionEntryPointInfo \*const defaultEntryPointInfo = functionBody->GetDefaultFunctionEntryPointInfo();

if(this == defaultEntryPointInfo)

{

if(simpleJitEntryPointInfo)

{

newEntryPoint = simpleJitEntryPointInfo;

functionBody->SetDefaultFunctionEntryPointInfo(

simpleJitEntryPointInfo,

reinterpret\_cast<JavascriptMethod>(newEntryPoint->GetNativeAddress()));

functionBody->SetExecutionMode(ExecutionMode::SimpleJit);

functionBody->ResetSimpleJitLimitAndCallCount();

}

#ifdef ASMJS\_PLAT

else if (functionBody->GetIsAsmJsFunction())

{

// the new entrypoint will be set to interpreter

newEntryPoint = functionBody->CreateNewDefaultEntryPoint();

newEntryPoint->SetIsAsmJSFunction(true);

newEntryPoint->address = AsmJsDefaultEntryThunk;

newEntryPoint->SetModuleAddress(GetModuleAddress());

functionBody->SetIsAsmJsFullJitScheduled(false);

functionBody->SetExecutionMode(functionBody->GetDefaultInterpreterExecutionMode());

this->functionProxy->SetOriginalEntryPoint(AsmJsDefaultEntryThunk);

}

#endif

else

{

newEntryPoint = functionBody->CreateNewDefaultEntryPoint();

functionBody->SetExecutionMode(functionBody->GetDefaultInterpreterExecutionMode());

}

functionBody->TraceExecutionMode("JitCodeExpired");

}

else

{

newEntryPoint = defaultEntryPointInfo;

}

OUTPUT\_TRACE(Js::ExpirableCollectPhase, L"Expiring 0x%p\n", this);

this->functionProxy->MapFunctionObjectTypes([&] (DynamicType\* type)

{

Assert(type->GetTypeId() == TypeIds\_Function);

ScriptFunctionType\* functionType = (ScriptFunctionType\*) type;

if (functionType->GetEntryPointInfo() == this)

{

OUTPUT\_TRACE(Js::ExpirableCollectPhase, L"Type 0x%p uses this entry point- switching to default entry point\n", this);

functionType->SetEntryPointInfo(newEntryPoint);

// we are allowed to replace the entry point on the type only if it's

// directly using the jitted code or a type is referencing this entry point

// but the entry point hasn't been called since the codegen thunk was installed on it

if (functionType->GetEntryPoint() == functionProxy->GetDirectEntryPoint(this) || IsIntermediateCodeGenThunk(functionType->GetEntryPoint()))

{

functionType->SetEntryPoint(this->functionProxy->GetDirectEntryPoint(newEntryPoint));

}

}

});

if(expiringSimpleJitEntryPointInfo)

{

// We could have just created a new entry point info that is using the simple JIT code. An allocation may have

// triggered shortly after, resulting in expiring the simple JIT entry point info. Update any entry point infos

// that are using the simple JIT code, and update the original entry point as necessary as well.

const JavascriptMethod newOriginalEntryPoint =

functionBody->GetDynamicInterpreterEntryPoint()

? static\_cast<JavascriptMethod>(

InterpreterThunkEmitter::ConvertToEntryPoint(functionBody->GetDynamicInterpreterEntryPoint()))

: DefaultEntryThunk;

const JavascriptMethod currentThunk = functionBody->GetScriptContext()->CurrentThunk;

const JavascriptMethod newDirectEntryPoint =

currentThunk == DefaultEntryThunk ? newOriginalEntryPoint : currentThunk;

const JavascriptMethod simpleJitNativeAddress = reinterpret\_cast<JavascriptMethod>(GetNativeAddress());

functionBody->MapEntryPoints([&](const int entryPointIndex, FunctionEntryPointInfo \*const entryPointInfo)

{

if(entryPointInfo != this && entryPointInfo->address == simpleJitNativeAddress)

{

entryPointInfo->address = newDirectEntryPoint;

}

});

if(functionBody->GetOriginalEntryPoint\_Unchecked() == simpleJitNativeAddress)

{

functionBody->SetOriginalEntryPoint(newOriginalEntryPoint);

functionBody->VerifyOriginalEntryPoint();

}

}

threadContext->QueueFreeOldEntryPointInfoIfInScript(this);

}

catch (Js::OutOfMemoryException)

{

// If we can't allocate a new entry point, skip expiring this object

if(expiringSimpleJitEntryPointInfo)

{

simpleJitEntryPointInfo = this;

functionBody->SetSimpleJitEntryPointInfo(this);

}

}

}

#endif

#ifdef PERF\_COUNTERS

void LoopEntryPointInfo::OnRecorded()

{

PERF\_COUNTER\_ADD(Code, TotalNativeCodeSize, GetCodeSize());

PERF\_COUNTER\_ADD(Code, LoopNativeCodeSize, GetCodeSize());

PERF\_COUNTER\_ADD(Code, DynamicNativeCodeSize, GetCodeSize());

}

#endif

FunctionBody \*LoopEntryPointInfo::GetFunctionBody() const

{

return loopHeader->functionBody;

}

//End AsmJs Support

void LoopEntryPointInfo::OnCleanup(bool isShutdown)

{

#ifndef TEMP\_DISABLE\_ASMJS

if (this->IsCodeGenDone() && !this->GetIsTJMode())

#else

if (this->IsCodeGenDone())

#endif

{

JS\_ETW(EtwTrace::LogLoopBodyUnloadEvent(this->loopHeader->functionBody, this->loopHeader, this));

#if ENABLE\_NATIVE\_CODEGEN

if (nullptr != this->inlineeFrameMap)

{

HeapDelete(this->inlineeFrameMap);

this->inlineeFrameMap = nullptr;

}

#endif

if (!isShutdown)

{

void\* currentCookie = nullptr;

ScriptContext\* scriptContext = this->loopHeader->functionBody->GetScriptContext();

#if ENABLE\_NATIVE\_CODEGEN

// In the debugger case, we might call cleanup after the native code gen that

// allocated this entry point has already shutdown. In that case, the validation

// check below should fail and we should not try to free this entry point

// since it's already been freed

NativeCodeGenerator\* currentNativeCodegen = scriptContext->GetNativeCodeGenerator();

Assert(this->validationCookie != nullptr);

currentCookie = (void\*)currentNativeCodegen;

#endif

if (validationCookie == currentCookie)

{

scriptContext->FreeLoopBody((Js::JavascriptMethod)this->GetNativeAddress());

}

}

#ifdef PERF\_COUNTERS

PERF\_COUNTER\_SUB(Code, TotalNativeCodeSize, GetCodeSize());

PERF\_COUNTER\_SUB(Code, LoopNativeCodeSize, GetCodeSize());

PERF\_COUNTER\_SUB(Code, DynamicNativeCodeSize, GetCodeSize());

#endif

}

}

#if ENABLE\_NATIVE\_CODEGEN

void LoopEntryPointInfo::OnNativeCodeInstallFailure()

{

this->ResetOnNativeCodeInstallFailure();

}

#endif

void LoopHeader::Init( FunctionBody \* functionBody )

{

// DisableJIT-TODO: Should this entire class be ifdefed out?

#if ENABLE\_NATIVE\_CODEGEN

this->functionBody = functionBody;

Recycler\* recycler = functionBody->GetScriptContext()->GetRecycler();

// Sync entryPoints changes to etw rundown lock

auto syncObj = functionBody->GetScriptContext()->GetThreadContext()->GetEtwRundownCriticalSection();

this->entryPoints = RecyclerNew(recycler, LoopEntryPointList, recycler, syncObj);

this->CreateEntryPoint();

#endif

}

#if ENABLE\_NATIVE\_CODEGEN

int LoopHeader::CreateEntryPoint()

{

ScriptContext\* scriptContext = this->functionBody->GetScriptContext();

Recycler\* recycler = scriptContext->GetRecycler();

LoopEntryPointInfo\* entryPoint = RecyclerNew(recycler, LoopEntryPointInfo, this, scriptContext->GetLibrary(), scriptContext->GetNativeCodeGenerator());

return this->entryPoints->Add(entryPoint);

}

void LoopHeader::ReleaseEntryPoints()

{

for (int iEntryPoint = 0; iEntryPoint < this->entryPoints->Count(); iEntryPoint++)

{

LoopEntryPointInfo \* entryPoint = this->entryPoints->Item(iEntryPoint);

if (entryPoint != nullptr && entryPoint->IsCodeGenDone())

{

// ReleaseEntryPoints is not called during recycler shutdown scenarios

// We also don't capture the cleanup stack since we've not seen cleanup bugs affect

// loop entry points so far. We can pass true here if this is no longer the case.

entryPoint->Cleanup(false /\* isShutdown \*/, false /\* capture cleanup stack \*/);

this->entryPoints->Item(iEntryPoint, nullptr);

}

}

}

#endif

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

void FunctionBody::DumpRegStats(FunctionBody \*funcBody)

{

if (funcBody->callCountStats == 0)

{

return;

}

uint loads = funcBody->regAllocLoadCount;

uint stores = funcBody->regAllocStoreCount;

if (Js::Configuration::Global.flags.NormalizeStats)

{

loads /= this->callCountStats;

stores /= this->callCountStats;

}

funcBody->DumpFullFunctionName();

Output::SkipToColumn(55);

Output::Print(L"Calls:%6d Loads:%9d Stores:%9d Total refs:%9d\n", this->callCountStats,

loads, stores, loads + stores);

}

#endif

Js::RegSlot FunctionBody::GetRestParamRegSlot()

{

Js::RegSlot dstRegSlot = GetConstantCount();

if (GetHasImplicitArgIns())

{

dstRegSlot += GetInParamsCount() - 1;

}

return dstRegSlot;

}

uint FunctionBody::GetNumberOfRecursiveCallSites()

{

uint recursiveInlineSpan = 0;

uint recursiveCallSiteInlineInfo = 0;

#if ENABLE\_PROFILE\_INFO

if (this->HasDynamicProfileInfo())

{

recursiveCallSiteInlineInfo = this->dynamicProfileInfo->GetRecursiveInlineInfo();

}

#endif

while (recursiveCallSiteInlineInfo)

{

recursiveInlineSpan += (recursiveCallSiteInlineInfo & 1);

recursiveCallSiteInlineInfo >>= 1;

}

return recursiveInlineSpan;

}

bool FunctionBody::CanInlineRecursively(uint depth, bool tryAggressive)

{

uint recursiveInlineSpan = this->GetNumberOfRecursiveCallSites();

uint minRecursiveInlineDepth = (uint)CONFIG\_FLAG(RecursiveInlineDepthMin);

if (recursiveInlineSpan != this->GetProfiledCallSiteCount() || tryAggressive == false)

{

return depth < minRecursiveInlineDepth;

}

uint maxRecursiveInlineDepth = (uint)CONFIG\_FLAG(RecursiveInlineDepthMax);

uint maxRecursiveBytecodeBudget = (uint)CONFIG\_FLAG(RecursiveInlineThreshold);

uint numberOfAllowedFuncs = maxRecursiveBytecodeBudget / this->m\_byteCodeWithoutLDACount;

uint maxDepth;

if (recursiveInlineSpan == 1)

{

maxDepth = numberOfAllowedFuncs;

}

else

{

maxDepth = (uint)ceil(log((double)((double)numberOfAllowedFuncs) / log((double)recursiveInlineSpan)));

}

maxDepth = maxDepth < minRecursiveInlineDepth ? minRecursiveInlineDepth : maxDepth;

maxDepth = maxDepth < maxRecursiveInlineDepth ? maxDepth : maxRecursiveInlineDepth;

return depth < maxDepth;

}

static const wchar\_t LoopWStr[] = L"Loop";

size\_t FunctionBody::GetLoopBodyName(uint loopNumber, \_Out\_writes\_opt\_z\_(size) wchar\_t\* nameBuffer, \_In\_ size\_t size)

{

const wchar\_t\* functionName = this->GetExternalDisplayName();

size\_t length = wcslen(functionName) + /\*length of largest int32\*/ 10 + \_countof(LoopWStr) + /\*null\*/ 1;

if (size < length || nameBuffer == nullptr)

{

return length;

}

int charsWritten = swprintf\_s(nameBuffer, length, L"%s%s%u", functionName, LoopWStr, loopNumber + 1);

Assert(charsWritten != -1);

return charsWritten + /\*nullptr\*/ 1;

}

}