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

// 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 Lowerer;

class Inline;

class FlowGraph;

#if defined(\_M\_ARM32\_OR\_ARM64)

#include "UnwindInfoManager.h"

#endif

struct Cloner

{

Cloner(Lowerer \*lowerer, JitArenaAllocator \*alloc) :

alloc(alloc),

symMap(nullptr),

labelMap(nullptr),

lowerer(lowerer),

instrFirst(nullptr),

instrLast(nullptr),

fRetargetClonedBranch(FALSE)

{

}

~Cloner()

{

if (symMap)

{

Adelete(alloc, symMap);

}

if (labelMap)

{

Adelete(alloc, labelMap);

}

}

void AddInstr(IR::Instr \* instrOrig, IR::Instr \* instrClone);

void Finish();

void RetargetClonedBranches();

HashTable<StackSym\*> \*symMap;

HashTable<IR::LabelInstr\*> \*labelMap;

Lowerer \* lowerer;

IR::Instr \* instrFirst;

IR::Instr \* instrLast;

BOOL fRetargetClonedBranch;

JitArenaAllocator \*alloc;

bool clonedInstrGetOrigArgSlotSym;

};

typedef JsUtil::Pair<uint32, IR::LabelInstr\*> YieldOffsetResumeLabel;

typedef JsUtil::List<YieldOffsetResumeLabel, JitArenaAllocator> YieldOffsetResumeLabelList;

typedef HashTable<uint32, JitArenaAllocator> SlotArrayCheckTable;

struct FrameDisplayCheckRecord

{

SlotArrayCheckTable \*table;

uint32 slotId;

FrameDisplayCheckRecord() : table(nullptr), slotId((uint32)-1) {}

};

typedef HashTable<FrameDisplayCheckRecord\*, JitArenaAllocator> FrameDisplayCheckTable;

class Func

{

public:

Func(JitArenaAllocator \*alloc, CodeGenWorkItem\* workItem, const Js::FunctionCodeGenRuntimeData \*const runtimeData,

Js::PolymorphicInlineCacheInfo \* const polymorphicInlineCacheInfo, CodeGenAllocators \*const codeGenAllocators,

CodeGenNumberAllocator \* numberAllocator, Js::ReadOnlyDynamicProfileInfo \*const profileInfo,

Js::ScriptContextProfiler \*const codeGenProfiler, const bool isBackgroundJIT, Func \* parentFunc = nullptr,

uint postCallByteCodeOffset = Js::Constants::NoByteCodeOffset,

Js::RegSlot returnValueRegSlot = Js::Constants::NoRegister, const bool isInlinedConstructor = false,

Js::ProfileId callSiteIdInParentFunc = UINT16\_MAX, bool isGetterSetter = false);

public:

ArenaAllocator \*GetCodeGenAllocator() const { return &this->m\_codeGenAllocators->allocator; }

CodeGenAllocators \* const GetCodeGenAllocators()

{

return this->GetTopFunc()->m\_codeGenAllocators;

}

NativeCodeData::Allocator \*GetNativeCodeDataAllocator()

{

return &this->GetTopFunc()->nativeCodeDataAllocator;

}

NativeCodeData::Allocator \*GetTransferDataAllocator()

{

return &this->GetTopFunc()->transferDataAllocator;

}

CodeGenNumberAllocator \* GetNumberAllocator()

{

return this->numberAllocator;

}

EmitBufferManager<CriticalSection> \*GetEmitBufferManager() const

{

return &this->m\_codeGenAllocators->emitBufferManager;

}

Js::ScriptContextProfiler \*GetCodeGenProfiler() const

{

#ifdef PROFILE\_EXEC

return m\_codeGenProfiler;

#else

return nullptr;

#endif

}

void InitLocalClosureSyms();

bool HasAnyStackNestedFunc() const { return this->hasAnyStackNestedFunc; }

bool DoStackNestedFunc() const { return this->stackNestedFunc; }

bool DoStackFrameDisplay() const { return this->stackClosure; }

bool DoStackScopeSlots() const { return this->stackClosure; }

bool IsBackgroundJIT() const { return this->m\_isBackgroundJIT; }

bool HasArgumentSlot() const { return this->GetInParamsCount() != 0 && !this->IsLoopBody(); }

bool IsLoopBody() const;

bool IsLoopBodyInTry() const;

bool CanAllocInPreReservedHeapPageSegment();

void SetDoFastPaths();

bool DoFastPaths() const { Assert(this->hasCalledSetDoFastPaths); return this->m\_doFastPaths; }

bool DoLoopFastPaths() const

{

return

(!IsSimpleJit() || Js::FunctionBody::IsNewSimpleJit()) &&

!PHASE\_OFF(Js::FastPathPhase, this) &&

!PHASE\_OFF(Js::LoopFastPathPhase, this);

}

bool DoGlobOpt() const

{

return

!PHASE\_OFF(Js::GlobOptPhase, this->GetJnFunction()) && !IsSimpleJit() &&

(!GetTopFunc()->HasTry() || GetTopFunc()->CanOptimizeTryCatch());

}

bool DoInline() const

{

return DoGlobOpt() && !GetTopFunc()->HasTry();

}

bool DoOptimizeTryCatch() const

{

Assert(IsTopFunc());

return DoGlobOpt();

}

bool CanOptimizeTryCatch() const

{

return !this->HasFinally() && !this->IsLoopBody() && !PHASE\_OFF(Js::OptimizeTryCatchPhase, this);

}

bool DoSimpleJitDynamicProfile() const { return IsSimpleJit() && GetTopFunc()->GetJnFunction()->DoSimpleJitDynamicProfile(); }

bool IsSimpleJit() const { return m\_workItem->GetJitMode() == ExecutionMode::SimpleJit; }

void BuildIR();

void Codegen();

void ThrowIfScriptClosed();

int32 StackAllocate(int size);

int32 StackAllocate(StackSym \*stackSym, int size);

void SetArgOffset(StackSym \*stackSym, int32 offset);

int32 GetLocalVarSlotOffset(int32 slotId);

int32 GetHasLocalVarChangedOffset();

bool IsJitInDebugMode();

bool IsNonTempLocalVar(uint32 slotIndex);

int32 AdjustOffsetValue(int32 offset);

void OnAddSym(Sym\* sym);

#ifdef MD\_GROW\_LOCALS\_AREA\_UP

void AjustLocalVarSlotOffset();

#endif

bool DoGlobOptsForGeneratorFunc();

static inline uint32 GetDiagLocalSlotSize()

{

// For the debug purpose we will have fixed stack slot size

// We will allocated the 8 bytes for each variable.

return MachDouble;

}

#ifdef DBG

// The pattern used to pre-fill locals for CHK builds.

// When we restore bailout values we check for this pattern, this is how we assert for non-initialized variables/garbage.

static const uint32 c\_debugFillPattern4 = 0xcececece;

static const unsigned \_\_int64 c\_debugFillPattern8 = 0xcececececececece;

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

static const uint32 c\_debugFillPattern = c\_debugFillPattern4;

#elif defined(\_M\_X64) || defined(\_M\_ARM64)

static const unsigned \_\_int64 c\_debugFillPattern = c\_debugFillPattern8;

#else

#error unsuported platform

#endif

#endif

uint32 GetInstrCount();

inline Js::ScriptContext\* GetScriptContext() const { return m\_workItem->GetScriptContext(); }

void NumberInstrs();

bool IsTopFunc() const { return this->parentFunc == nullptr; }

Func const \* GetTopFunc() const;

Func \* GetTopFunc();

void SetFirstArgOffset(IR::Instr\* inlineeStart);

uint GetFunctionNumber() const

{

Assert(this->IsTopFunc());

return this->m\_workItem->GetFunctionNumber();

}

uint GetLocalFunctionId() const

{

return this->m\_workItem->GetFunctionBody()->GetLocalFunctionId();

}

uint GetSourceContextId() const

{

return this->m\_workItem->GetFunctionBody()->GetSourceContextId();

}

BOOL HasTry() const

{

Assert(this->IsTopFunc());

Assert(this->m\_jnFunction); // For now we always have a function body

return this->m\_jnFunction->GetHasTry();

}

bool HasFinally() const

{

Assert(this->IsTopFunc());

Assert(this->m\_jnFunction); // For now we always have a function body

return this->m\_jnFunction->GetHasFinally();

}

Js::ArgSlot GetInParamsCount() const

{

Assert(this->IsTopFunc());

Assert(this->m\_jnFunction); // For now we always have a function body

return this->m\_jnFunction->GetInParamsCount();

}

bool IsGlobalFunc() const

{

Assert(this->IsTopFunc());

Assert(this->m\_jnFunction); // For now we always have a function body

return this->m\_jnFunction->GetIsGlobalFunc();

}

RecyclerWeakReference<Js::FunctionBody> \*GetWeakFuncRef() const;

Js::FunctionBody \* GetJnFunction() const { return m\_jnFunction; }

StackSym \*EnsureLoopParamSym();

StackSym \*GetFuncObjSym() const { return m\_funcObjSym; }

void SetFuncObjSym(StackSym \*sym) { m\_funcObjSym = sym; }

StackSym \*GetJavascriptLibrarySym() const { return m\_javascriptLibrarySym; }

void SetJavascriptLibrarySym(StackSym \*sym) { m\_javascriptLibrarySym = sym; }

StackSym \*GetScriptContextSym() const { return m\_scriptContextSym; }

void SetScriptContextSym(StackSym \*sym) { m\_scriptContextSym = sym; }

StackSym \*GetFunctionBodySym() const { return m\_functionBodySym; }

void SetFunctionBodySym(StackSym \*sym) { m\_functionBodySym = sym; }

StackSym \*GetLocalClosureSym() const { return m\_localClosureSym; }

void SetLocalClosureSym(StackSym \*sym) { m\_localClosureSym = sym; }

StackSym \*GetLocalFrameDisplaySym() const { return m\_localFrameDisplaySym; }

void SetLocalFrameDisplaySym(StackSym \*sym) { m\_localFrameDisplaySym = sym; }

uint8 \*GetCallsCountAddress() const;

void EnsurePinnedTypeRefs();

void PinTypeRef(void\* typeRef);

void EnsureSingleTypeGuards();

Js::JitTypePropertyGuard\* GetOrCreateSingleTypeGuard(Js::Type\* type);

void EnsureEquivalentTypeGuards();

Js::JitEquivalentTypeGuard \* CreateEquivalentTypeGuard(Js::Type\* type, uint32 objTypeSpecFldId);

void EnsurePropertyGuardsByPropertyId();

void EnsureCtorCachesByPropertyId();

void LinkGuardToPropertyId(Js::PropertyId propertyId, Js::JitIndexedPropertyGuard\* guard);

void LinkCtorCacheToPropertyId(Js::PropertyId propertyId, Js::JitTimeConstructorCache\* cache);

Js::JitTimeConstructorCache\* GetConstructorCache(const Js::ProfileId profiledCallSiteId);

void SetConstructorCache(const Js::ProfileId profiledCallSiteId, Js::JitTimeConstructorCache\* constructorCache);

void EnsurePropertiesWrittenTo();

void EnsureCallSiteToArgumentsOffsetFixupMap();

IR::LabelInstr \* EnsureFuncStartLabel();

IR::LabelInstr \* GetFuncStartLabel();

IR::LabelInstr \* EnsureFuncEndLabel();

IR::LabelInstr \* GetFuncEndLabel();

#ifdef \_M\_X64

void SetSpillSize(int32 spillSize)

{

m\_spillSize = spillSize;

}

int32 GetSpillSize()

{

return m\_spillSize;

}

void SetArgsSize(int32 argsSize)

{

m\_argsSize = argsSize;

}

int32 GetArgsSize()

{

return m\_argsSize;

}

void SetSavedRegSize(int32 savedRegSize)

{

m\_savedRegSize = savedRegSize;

}

int32 GetSavedRegSize()

{

return m\_savedRegSize;

}

#endif

bool IsInlinee() const

{

Assert(m\_inlineeFrameStartSym ? (m\_inlineeFrameStartSym->m\_offset != -1) : true);

return m\_inlineeFrameStartSym != nullptr;

}

void SetInlineeFrameStartSym(StackSym \*sym)

{

Assert(m\_inlineeFrameStartSym == nullptr);

m\_inlineeFrameStartSym = sym;

}

IR::SymOpnd \*GetInlineeArgCountSlotOpnd()

{

return GetInlineeOpndAtOffset(Js::Constants::InlineeMetaArgIndex\_Argc \* MachPtr);

}

IR::SymOpnd \*GetNextInlineeFrameArgCountSlotOpnd()

{

Assert(!this->m\_hasInlineArgsOpt);

return GetInlineeOpndAtOffset((Js::Constants::InlineeMetaArgCount + actualCount) \* MachPtr);

}

IR::SymOpnd \*GetInlineeFunctionObjectSlotOpnd()

{

Assert(!this->m\_hasInlineArgsOpt);

return GetInlineeOpndAtOffset(Js::Constants::InlineeMetaArgIndex\_FunctionObject \* MachPtr);

}

IR::SymOpnd \*GetInlineeArgumentsObjectSlotOpnd()

{

return GetInlineeOpndAtOffset(Js::Constants::InlineeMetaArgIndex\_ArgumentsObject \* MachPtr);

}

IR::SymOpnd \*GetInlineeArgvSlotOpnd()

{

Assert(!this->m\_hasInlineArgsOpt);

return GetInlineeOpndAtOffset(Js::Constants::InlineeMetaArgIndex\_Argv \* MachPtr);

}

bool IsInlined() const

{

return this->parentFunc != nullptr;

}

bool IsInlinedConstructor() const

{

return this->isInlinedConstructor;

}

bool IsTJLoopBody()const {

return this->isTJLoopBody;

}

Js::ObjTypeSpecFldInfo\* GetObjTypeSpecFldInfo(const uint index) const;

Js::ObjTypeSpecFldInfo\* GetGlobalObjTypeSpecFldInfo(uint propertyInfoId) const;

void SetGlobalObjTypeSpecFldInfo(uint propertyInfoId, Js::ObjTypeSpecFldInfo\* info);

// Gets an inline cache pointer to use in jitted code. Cached data may not be stable while jitting. Does not return null.

Js::InlineCache \*GetRuntimeInlineCache(const uint index) const;

Js::PolymorphicInlineCache \* GetRuntimePolymorphicInlineCache(const uint index) const;

byte GetPolyCacheUtil(const uint index) const;

byte GetPolyCacheUtilToInitialize(const uint index) const;

#if defined(\_M\_ARM32\_OR\_ARM64)

RegNum GetLocalsPointer() const;

#endif

#if DBG\_DUMP

void Dump(IRDumpFlags flags);

void Dump();

void DumpHeader();

#endif

#if DBG\_DUMP || defined(ENABLE\_IR\_VIEWER)

LPCSTR GetVtableName(INT\_PTR address);

#endif

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

bool DoRecordNativeMap() const;

#endif

public:

JitArenaAllocator \* m\_alloc;

CodeGenWorkItem\* m\_workItem;

const Js::FunctionCodeGenJitTimeData \*const m\_jitTimeData;

const Js::FunctionCodeGenRuntimeData \*const m\_runtimeData;

Js::PolymorphicInlineCacheInfo \*const m\_polymorphicInlineCacheInfo;

// This indicates how many constructor caches we inserted into the constructorCaches array, not the total size of the array.

uint constructorCacheCount;

// This array maps callsite ids to constructor caches. The size corresponds to the number of callsites in the function.

Js::JitTimeConstructorCache\*\* constructorCaches;

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

TypeRefSet\* pinnedTypeRefs;

typedef JsUtil::BaseDictionary<Js::Type\*, Js::JitTypePropertyGuard\*, JitArenaAllocator, PowerOf2SizePolicy> TypePropertyGuardDictionary;

TypePropertyGuardDictionary\* singleTypeGuards;

typedef SListCounted<Js::JitEquivalentTypeGuard\*> EquivalentTypeGuardList;

EquivalentTypeGuardList\* equivalentTypeGuards;

typedef JsUtil::BaseHashSet<Js::JitIndexedPropertyGuard\*, JitArenaAllocator, PowerOf2SizePolicy> IndexedPropertyGuardSet;

typedef JsUtil::BaseDictionary<Js::PropertyId, IndexedPropertyGuardSet\*, JitArenaAllocator, PowerOf2SizePolicy> PropertyGuardByPropertyIdMap;

PropertyGuardByPropertyIdMap\* propertyGuardsByPropertyId;

typedef JsUtil::BaseHashSet<Js::ConstructorCache\*, JitArenaAllocator, PowerOf2SizePolicy> CtorCacheSet;

typedef JsUtil::BaseDictionary<Js::PropertyId, CtorCacheSet\*, JitArenaAllocator, PowerOf2SizePolicy> CtorCachesByPropertyIdMap;

CtorCachesByPropertyIdMap\* ctorCachesByPropertyId;

typedef JsUtil::BaseDictionary<Js::ProfileId, int32, JitArenaAllocator, PrimeSizePolicy> CallSiteToArgumentsOffsetFixupMap;

CallSiteToArgumentsOffsetFixupMap\* callSiteToArgumentsOffsetFixupMap;

int indexedPropertyGuardCount;

typedef JsUtil::BaseHashSet<Js::PropertyId, JitArenaAllocator> PropertyIdSet;

PropertyIdSet\* propertiesWrittenTo;

PropertyIdSet lazyBailoutProperties;

bool anyPropertyMayBeWrittenTo;

SlotArrayCheckTable \*slotArrayCheckTable;

FrameDisplayCheckTable \*frameDisplayCheckTable;

IR::Instr \* m\_headInstr;

IR::Instr \* m\_exitInstr;

IR::Instr \* m\_tailInstr;

#ifdef \_M\_X64

int32 m\_spillSize;

int32 m\_argsSize;

int32 m\_savedRegSize;

PrologEncoder m\_prologEncoder;

#endif

SymTable \* m\_symTable;

StackSym \* m\_loopParamSym;

StackSym \* m\_funcObjSym;

StackSym \* m\_javascriptLibrarySym;

StackSym \* m\_scriptContextSym;

StackSym \* m\_functionBodySym;

StackSym \* m\_localClosureSym;

StackSym \* m\_localFrameDisplaySym;

StackSym \* m\_bailoutReturnValueSym;

StackSym \* m\_hasBailedOutSym;

int32 m\_localStackHeight;

uint frameSize;

uint32 inlineDepth;

uint32 postCallByteCodeOffset;

Js::RegSlot returnValueRegSlot;

Js::ArgSlot actualCount;

int32 firstActualStackOffset;

uint32 tryCatchNestingLevel;

uint32 m\_totalJumpTableSizeInBytesForSwitchStatements;

#if defined(\_M\_ARM32\_OR\_ARM64)

//Offset to arguments from sp + m\_localStackHeight;

//For non leaf functions this is (callee saved register count + LR + R11) \* MachRegInt

//For leaf functions this is (saved registers) \* MachRegInt

int32 m\_ArgumentsOffset;

UnwindInfoManager m\_unwindInfo;

IR::LabelInstr \* m\_epilogLabel;

#endif

IR::LabelInstr \* m\_funcStartLabel;

IR::LabelInstr \* m\_funcEndLabel;

// Keep track of the maximum number of args on the stack.

uint32 m\_argSlotsForFunctionsCalled;

#if DBG

uint32 m\_callSiteCount;

#endif

FlowGraph \* m\_fg;

unsigned int m\_labelCount;

BitVector m\_regsUsed;

StackSym \* tempSymDouble;

uint32 loopCount;

Js::ProfileId callSiteIdInParentFunc;

bool m\_isLeaf: 1; // This is set in the IRBuilder and might be inaccurate after inlining

bool m\_hasCalls: 1; // This is more accurate compared to m\_isLeaf

bool m\_hasInlineArgsOpt : 1;

bool m\_doFastPaths : 1;

bool hasBailout: 1;

bool hasBailoutInEHRegion : 1;

bool hasStackArgs: 1;

bool hasArgumentObject : 1;

bool hasUnoptimizedArgumentsAcccess : 1; // True if there are any arguments access beyond the simple case of this.apply pattern

bool m\_canDoInlineArgsOpt : 1;

bool hasApplyTargetInlining:1;

bool isGetterSetter : 1;

const bool isInlinedConstructor: 1;

bool hasImplicitCalls: 1;

bool hasTempObjectProducingInstr:1; // At least one instruction which can produce temp object

bool isTJLoopBody : 1;

bool isFlowGraphValid : 1;

#if DBG

bool hasCalledSetDoFastPaths:1;

bool isPostLower:1;

bool isPostRegAlloc:1;

bool isPostPeeps:1;

bool isPostLayout:1;

bool isPostFinalLower:1;

typedef JsUtil::Stack<Js::Phase> CurrentPhasesStack;

CurrentPhasesStack currentPhases;

bool IsInPhase(Js::Phase tag);

#endif

void BeginPhase(Js::Phase tag);

void EndPhase(Js::Phase tag, bool dump = true);

void EndProfiler(Js::Phase tag);

void BeginClone(Lowerer \*lowerer, JitArenaAllocator \*alloc);

void EndClone();

Cloner \* GetCloner() const { return GetTopFunc()->m\_cloner; }

InstrMap \* GetCloneMap() const { return GetTopFunc()->m\_cloneMap; }

void ClearCloneMap() { Assert(this->IsTopFunc()); this->m\_cloneMap = nullptr; }

bool HasByteCodeOffset() const { return !this->GetTopFunc()->hasInstrNumber; }

bool DoMaintainByteCodeOffset() const { return this->HasByteCodeOffset() && this->GetTopFunc()->maintainByteCodeOffset; }

void StopMaintainByteCodeOffset() { this->GetTopFunc()->maintainByteCodeOffset = false; }

Func \* GetParentFunc() const { return parentFunc; }

uint GetMaxInlineeArgOutCount() const { return maxInlineeArgOutCount; }

void UpdateMaxInlineeArgOutCount(uint inlineeArgOutCount);

#if DBG\_DUMP

ptrdiff\_t m\_codeSize;

#endif

bool GetHasCalls() const { return this->m\_hasCalls; }

void SetHasCalls() { this->m\_hasCalls = true; }

void SetHasCallsOnSelfAndParents()

{

Func \*curFunc = this;

while (curFunc)

{

curFunc->SetHasCalls();

curFunc = curFunc->GetParentFunc();

}

}

void SetHasInstrNumber(bool has) { this->GetTopFunc()->hasInstrNumber = has; }

bool HasInstrNumber() const { return this->GetTopFunc()->hasInstrNumber; }

bool HasInlinee() const { Assert(this->IsTopFunc()); return this->hasInlinee; }

void SetHasInlinee() { Assert(this->IsTopFunc()); this->hasInlinee = true; }

bool GetThisOrParentInlinerHasArguments() const { return thisOrParentInlinerHasArguments; }

bool GetHasStackArgs() const { return this->hasStackArgs;}

void SetHasStackArgs(bool has) { this->hasStackArgs = has;}

bool GetHasArgumentObject() const { return this->hasArgumentObject;}

void SetHasArgumentObject() { this->hasArgumentObject = true;}

bool GetHasUnoptimizedArgumentsAcccess() const { return this->hasUnoptimizedArgumentsAcccess; }

void SetHasUnoptimizedArgumentsAccess(bool args)

{

// Once set to 'true' make sure this does not become false

if (!this->hasUnoptimizedArgumentsAcccess)

{

this->hasUnoptimizedArgumentsAcccess = args;

}

if (args)

{

Func \*curFunc = this->GetParentFunc();

while (curFunc)

{

curFunc->hasUnoptimizedArgumentsAcccess = args;

curFunc = curFunc->GetParentFunc();

}

}

}

void DisableCanDoInlineArgOpt()

{

Func\* curFunc = this;

while (curFunc)

{

curFunc->m\_canDoInlineArgsOpt = false;

curFunc->m\_hasInlineArgsOpt = false;

curFunc = curFunc->GetParentFunc();

}

}

bool GetHasApplyTargetInlining() const { return this->hasApplyTargetInlining;}

void SetHasApplyTargetInlining() { this->hasApplyTargetInlining = true;}

bool GetHasMarkTempObjects() const { return this->hasMarkTempObjects; }

void SetHasMarkTempObjects() { this->hasMarkTempObjects = true; }

bool GetHasImplicitCalls() const { return this->hasImplicitCalls;}

void SetHasImplicitCalls(bool has) { this->hasImplicitCalls = has;}

void SetHasImplicitCallsOnSelfAndParents()

{

this->SetHasImplicitCalls(true);

Func \*curFunc = this->GetParentFunc();

while (curFunc && !curFunc->IsTopFunc())

{

curFunc->SetHasImplicitCalls(true);

curFunc = curFunc->GetParentFunc();

}

}

bool GetHasTempObjectProducingInstr() const { return this->hasTempObjectProducingInstr; }

void SetHasTempObjectProducingInstr(bool has) { this->hasTempObjectProducingInstr = has; }

Js::ReadOnlyDynamicProfileInfo \* GetProfileInfo() const { return this->profileInfo; }

bool HasProfileInfo() { return this->profileInfo->HasProfileInfo(); }

bool HasArrayInfo()

{

const auto top = this->GetTopFunc();

return this->HasProfileInfo() && this->GetWeakFuncRef() && !(top->HasTry() && !top->DoOptimizeTryCatch()) &&

top->DoGlobOpt() && !PHASE\_OFF(Js::LoopFastPathPhase, top);

}

static Js::BuiltinFunction GetBuiltInIndex(IR::Opnd\* opnd)

{

Assert(opnd);

Js::BuiltinFunction index;

if (opnd->IsRegOpnd())

{

index = opnd->AsRegOpnd()->m\_sym->m\_builtInIndex;

}

else if (opnd->IsSymOpnd())

{

PropertySym \*propertySym = opnd->AsSymOpnd()->m\_sym->AsPropertySym();

index = Js::JavascriptLibrary::GetBuiltinFunctionForPropId(propertySym->m\_propertyId);

}

else

{

index = Js::BuiltinFunction::None;

}

return index;

}

static bool IsBuiltInInlinedInLowerer(IR::Opnd\* opnd)

{

Assert(opnd);

Js::BuiltinFunction index = Func::GetBuiltInIndex(opnd);

switch (index)

{

case Js::BuiltinFunction::String\_CharAt:

case Js::BuiltinFunction::String\_CharCodeAt:

case Js::BuiltinFunction::String\_CodePointAt:

case Js::BuiltinFunction::Math\_Abs:

case Js::BuiltinFunction::Array\_Push:

case Js::BuiltinFunction::String\_Replace:

return true;

default:

return false;

}

}

void AddYieldOffsetResumeLabel(uint32 offset, IR::LabelInstr\* label)

{

m\_yieldOffsetResumeLabelList->Add(YieldOffsetResumeLabel(offset, label));

}

template <typename Fn>

void MapYieldOffsetResumeLabels(Fn fn)

{

m\_yieldOffsetResumeLabelList->Map(fn);

}

template <typename Fn>

bool MapUntilYieldOffsetResumeLabels(Fn fn)

{

return m\_yieldOffsetResumeLabelList->MapUntil(fn);

}

void RemoveYieldOffsetResumeLabel(const YieldOffsetResumeLabel& yorl)

{

m\_yieldOffsetResumeLabelList->Remove(yorl);

}

void RemoveDeadYieldOffsetResumeLabel(IR::LabelInstr\* label)

{

uint32 offset;

bool found = m\_yieldOffsetResumeLabelList->MapUntil([&offset, &label](int i, YieldOffsetResumeLabel& yorl)

{

if (yorl.Second() == label)

{

offset = yorl.First();

return true;

}

return false;

});

Assert(found);

RemoveYieldOffsetResumeLabel(YieldOffsetResumeLabel(offset, label));

AddYieldOffsetResumeLabel(offset, nullptr);

}

IR::Instr \* GetFunctionEntryInsertionPoint();

IR::IndirOpnd \* GetConstantAddressIndirOpnd(void \* address, IR::AddrOpndKind kind, IRType type, Js::OpCode loadOpCode);

void MarkConstantAddressSyms(BVSparse<JitArenaAllocator> \* bv);

void DisableConstandAddressLoadHoist() { canHoistConstantAddressLoad = false; }

void AddSlotArrayCheck(IR::SymOpnd \*fieldOpnd);

void AddFrameDisplayCheck(IR::SymOpnd \*fieldOpnd, uint32 slotId = (uint32)-1);

#if DBG

bool allowRemoveBailOutArgInstr;

#endif

#if defined(\_M\_ARM32\_OR\_ARM64)

int32 GetInlineeArgumentStackSize()

{

int32 count = this->GetMaxInlineeArgOutCount();

if (count)

{

return ((count + 1) \* MachPtr); // +1 for the dedicated zero out argc slot

}

return 0;

}

#endif

public:

BVSparse<JitArenaAllocator> \* argObjSyms;

BVSparse<JitArenaAllocator> \* m\_nonTempLocalVars; // Only populated in debug mode as part of IRBuilder. Used in GlobOpt and BackwardPass.

InlineeFrameInfo\* frameInfo;

uint32 m\_inlineeId;

IR::LabelInstr \* m\_bailOutNoSaveLabel;

private:

#ifdef PROFILE\_EXEC

Js::ScriptContextProfiler \*const m\_codeGenProfiler;

#endif

Js::FunctionBody\* m\_jnFunction;

Func \* const parentFunc;

StackSym \* m\_inlineeFrameStartSym;

uint maxInlineeArgOutCount;

const bool m\_isBackgroundJIT;

bool hasInstrNumber;

bool maintainByteCodeOffset;

bool hasInlinee;

bool thisOrParentInlinerHasArguments;

bool useRuntimeStats;

bool stackNestedFunc;

bool stackClosure;

bool hasAnyStackNestedFunc;

bool hasMarkTempObjects;

Cloner \* m\_cloner;

InstrMap \* m\_cloneMap;

Js::ReadOnlyDynamicProfileInfo \*const profileInfo;

NativeCodeData::Allocator nativeCodeDataAllocator;

NativeCodeData::Allocator transferDataAllocator;

CodeGenNumberAllocator \* numberAllocator;

int32 m\_localVarSlotsOffset;

int32 m\_hasLocalVarChangedOffset; // Offset on stack of 1 byte which indicates if any local var has changed.

CodeGenAllocators \*const m\_codeGenAllocators;

YieldOffsetResumeLabelList \* m\_yieldOffsetResumeLabelList;

StackSym \*CreateInlineeStackSym();

IR::SymOpnd \*GetInlineeOpndAtOffset(int32 offset);

bool HasLocalVarSlotCreated() const { return m\_localVarSlotsOffset != Js::Constants::InvalidOffset; }

void EnsureLocalVarSlots();

SList<IR::RegOpnd \*> constantAddressRegOpnd;

IR::Instr \* lastConstantAddressRegLoadInstr;

bool canHoistConstantAddressLoad;

#if DBG

VtableHashMap \* vtableMap;

#endif

};

class AutoCodeGenPhase

{

public:

AutoCodeGenPhase(Func \* func, Js::Phase phase) : func(func), phase(phase), dump(false), isPhaseComplete(false)

{

func->BeginPhase(phase);

}

~AutoCodeGenPhase()

{

if(this->isPhaseComplete)

{

func->EndPhase(phase, dump);

}

else

{

//End the profiler tag

func->EndProfiler(phase);

}

}

void EndPhase(Func \* func, Js::Phase phase, bool dump, bool isPhaseComplete)

{

Assert(this->func == func);

Assert(this->phase == phase);

this->dump = dump && (PHASE\_DUMP(Js::SimpleJitPhase, func->GetJnFunction()) || !func->IsSimpleJit());

this->isPhaseComplete = isPhaseComplete;

}

private:

Func \* func;

Js::Phase phase;

bool dump;

bool isPhaseComplete;

};

#define BEGIN\_CODEGEN\_PHASE(func, phase) { AutoCodeGenPhase \_\_autoCodeGen(func, phase);

#define END\_CODEGEN\_PHASE(func, phase) \_\_autoCodeGen.EndPhase(func, phase, true, true); }

#define END\_CODEGEN\_PHASE\_NO\_DUMP(func, phase) \_\_autoCodeGen.EndPhase(func, phase, false, true); }

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

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

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

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

#include "BackEnd.h"

#include "Base\EtwTrace.h"

#include "Base\ScriptContextProfiler.h"

Func::Func(JitArenaAllocator \*alloc, CodeGenWorkItem\* workItem, const Js::FunctionCodeGenRuntimeData \*const runtimeData,

Js::PolymorphicInlineCacheInfo \* const polymorphicInlineCacheInfo, CodeGenAllocators \*const codeGenAllocators,

CodeGenNumberAllocator \* numberAllocator, Js::ReadOnlyDynamicProfileInfo \*const profileInfo,

Js::ScriptContextProfiler \*const codeGenProfiler, const bool isBackgroundJIT, Func \* parentFunc,

uint postCallByteCodeOffset, Js::RegSlot returnValueRegSlot, const bool isInlinedConstructor,

Js::ProfileId callSiteIdInParentFunc, bool isGetterSetter) :

m\_alloc(alloc),

m\_workItem(workItem),

m\_jitTimeData(workItem->RecyclableData()->JitTimeData()),

m\_runtimeData(runtimeData),

m\_polymorphicInlineCacheInfo(polymorphicInlineCacheInfo),

m\_codeGenAllocators(codeGenAllocators),

m\_inlineeId(0),

pinnedTypeRefs(nullptr),

singleTypeGuards(nullptr),

equivalentTypeGuards(nullptr),

propertyGuardsByPropertyId(nullptr),

ctorCachesByPropertyId(nullptr),

callSiteToArgumentsOffsetFixupMap(nullptr),

indexedPropertyGuardCount(0),

propertiesWrittenTo(nullptr),

lazyBailoutProperties(alloc),

anyPropertyMayBeWrittenTo(false),

#ifdef PROFILE\_EXEC

m\_codeGenProfiler(codeGenProfiler),

#endif

m\_isBackgroundJIT(isBackgroundJIT),

m\_cloner(nullptr),

m\_cloneMap(nullptr),

m\_loopParamSym(nullptr),

m\_funcObjSym(nullptr),

m\_localClosureSym(nullptr),

m\_localFrameDisplaySym(nullptr),

m\_bailoutReturnValueSym(nullptr),

m\_hasBailedOutSym(nullptr),

m\_inlineeFrameStartSym(nullptr),

m\_regsUsed(0),

m\_fg(nullptr),

m\_labelCount(0),

m\_argSlotsForFunctionsCalled(0),

m\_isLeaf(false),

m\_hasCalls(false),

m\_hasInlineArgsOpt(false),

m\_canDoInlineArgsOpt(true),

m\_doFastPaths(false),

hasBailout(false),

hasBailoutInEHRegion(false),

hasInstrNumber(false),

maintainByteCodeOffset(true),

frameSize(0),

parentFunc(parentFunc),

argObjSyms(nullptr),

m\_nonTempLocalVars(nullptr),

hasAnyStackNestedFunc(false),

hasMarkTempObjects(false),

postCallByteCodeOffset(postCallByteCodeOffset),

maxInlineeArgOutCount(0),

returnValueRegSlot(returnValueRegSlot),

firstActualStackOffset(-1),

m\_localVarSlotsOffset(Js::Constants::InvalidOffset),

m\_hasLocalVarChangedOffset(Js::Constants::InvalidOffset),

actualCount((Js::ArgSlot) - 1),

tryCatchNestingLevel(0),

m\_localStackHeight(0),

tempSymDouble(nullptr),

hasInlinee(false),

thisOrParentInlinerHasArguments(false),

hasStackArgs(false),

hasArgumentObject(false),

hasUnoptimizedArgumentsAcccess(false),

hasApplyTargetInlining(false),

hasImplicitCalls(false),

hasTempObjectProducingInstr(false),

isInlinedConstructor(isInlinedConstructor),

numberAllocator(numberAllocator),

profileInfo(profileInfo),

loopCount(0),

callSiteIdInParentFunc(callSiteIdInParentFunc),

isGetterSetter(isGetterSetter),

frameInfo(nullptr),

isTJLoopBody(false),

isFlowGraphValid(false),

#if DBG

m\_callSiteCount(0),

#endif

stackNestedFunc(false),

stackClosure(false)

#if defined(\_M\_ARM32\_OR\_ARM64)

, m\_ArgumentsOffset(0)

, m\_epilogLabel(nullptr)

#endif

, m\_funcStartLabel(nullptr)

, m\_funcEndLabel(nullptr)

#ifdef \_M\_X64

, m\_prologEncoder(alloc)

#endif

#if DBG

, hasCalledSetDoFastPaths(false)

, allowRemoveBailOutArgInstr(false)

, currentPhases(alloc)

, isPostLower(false)

, isPostRegAlloc(false)

, isPostPeeps(false)

, isPostLayout(false)

, isPostFinalLower(false)

, vtableMap(nullptr)

#endif

, m\_yieldOffsetResumeLabelList(nullptr)

, m\_bailOutNoSaveLabel(nullptr)

, constantAddressRegOpnd(alloc)

, lastConstantAddressRegLoadInstr(nullptr)

, m\_totalJumpTableSizeInBytesForSwitchStatements(0)

, slotArrayCheckTable(nullptr)

, frameDisplayCheckTable(nullptr)

{

Assert(this->IsInlined() == !!runtimeData);

if (this->IsInlined())

{

m\_inlineeId = ++(GetTopFunc()->m\_inlineeId);

}

m\_jnFunction = m\_workItem->GetFunctionBody();

bool doStackNestedFunc = m\_jnFunction->DoStackNestedFunc();

bool doStackClosure = m\_jnFunction->DoStackClosure() && !PHASE\_OFF(Js::FrameDisplayFastPathPhase, this);

Assert(!doStackClosure || doStackNestedFunc);

this->stackClosure = doStackClosure && this->IsTopFunc();

if (this->stackClosure)

{

m\_workItem->GetEntryPoint()->SetHasJittedStackClosure();

}

if (m\_jnFunction->GetDoBackendArgumentsOptimization() && !m\_jnFunction->GetHasTry())

{

// doBackendArgumentsOptimization bit is set when there is no eval inside a function

// as determined by the bytecode generator.

SetHasStackArgs(true);

}

if (m\_workItem->Type() == JsFunctionType)

{

if (doStackNestedFunc && m\_jnFunction->GetNestedCount() != 0)

{

Assert(!(this->IsJitInDebugMode() && !m\_jnFunction->GetUtf8SourceInfo()->GetIsLibraryCode()));

stackNestedFunc = true;

this->GetTopFunc()->hasAnyStackNestedFunc = true;

}

}

else

{

Assert(m\_workItem->Type() == JsLoopBodyWorkItemType);

}

if (m\_jnFunction->GetHasOrParentHasArguments() || parentFunc && parentFunc->thisOrParentInlinerHasArguments)

{

thisOrParentInlinerHasArguments = true;

}

if (parentFunc == nullptr)

{

inlineDepth = 0;

m\_symTable = JitAnew(alloc, SymTable);

m\_symTable->Init(this);

Assert(Js::Constants::NoByteCodeOffset == postCallByteCodeOffset);

Assert(Js::Constants::NoRegister == returnValueRegSlot);

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

if (HasArgumentSlot())

{

// Pre-allocate the single argument slot we'll reserve for the arguments object.

// For ARM, the argument slot is not part of the local but part of the register saves

m\_localStackHeight = MachArgsSlotOffset;

}

#endif

}

else

{

inlineDepth = parentFunc->inlineDepth + 1;

Assert(Js::Constants::NoByteCodeOffset != postCallByteCodeOffset);

}

this->constructorCacheCount = 0;

this->constructorCaches = AnewArrayZ(this->m\_alloc, Js::JitTimeConstructorCache\*, this->m\_jnFunction->GetProfiledCallSiteCount());

#if DBG\_DUMP

m\_codeSize = -1;

#endif

#if defined(\_M\_X64)

m\_spillSize = -1;

m\_argsSize = -1;

m\_savedRegSize = -1;

#endif

if (this->IsJitInDebugMode())

{

m\_nonTempLocalVars = Anew(this->m\_alloc, BVSparse<JitArenaAllocator>, this->m\_alloc);

}

if (this->m\_jnFunction->IsGenerator())

{

m\_yieldOffsetResumeLabelList = YieldOffsetResumeLabelList::New(this->m\_alloc);

}

canHoistConstantAddressLoad = !PHASE\_OFF(Js::HoistConstAddrPhase, this);

}

bool

Func::IsLoopBody() const

{

return this->m\_workItem->Type() == JsLoopBodyWorkItemType;

}

bool

Func::IsLoopBodyInTry() const

{

return IsLoopBody() && ((JsLoopBodyCodeGen\*)this->m\_workItem)->loopHeader->isInTry;

}

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

///

/// Func::Codegen

///

/// Codegen this function.

///

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

void

Func::Codegen()

{

Assert(!IsJitInDebugMode() || !m\_jnFunction->GetHasTry());

Js::ScriptContext\* scriptContext = this->GetScriptContext();

{

if(IS\_JS\_ETW(EventEnabledJSCRIPT\_FUNCTION\_JIT\_START()))

{

WCHAR displayNameBuffer[256];

WCHAR\* displayName = displayNameBuffer;

size\_t sizeInChars = this->m\_workItem->GetDisplayName(displayName, 256);

if(sizeInChars > 256)

{

displayName = new WCHAR[sizeInChars];

this->m\_workItem->GetDisplayName(displayName, 256);

}

JS\_ETW(EventWriteJSCRIPT\_FUNCTION\_JIT\_START(

this->GetFunctionNumber(),

displayName,

this->GetScriptContext(),

this->m\_workItem->GetInterpretedCount(),

(const unsigned int)this->m\_jnFunction->LengthInBytes(),

this->m\_jnFunction->GetByteCodeCount(),

this->m\_jnFunction->GetByteCodeInLoopCount(),

(int)this->m\_workItem->GetJitMode()));

if(displayName != displayNameBuffer)

{

delete[] displayName;

}

}

}

#if DBG\_DUMP

if (Js::Configuration::Global.flags.TestTrace.IsEnabled(Js::BackEndPhase))

{

if (this->IsLoopBody())

{

Output::Print(L"---BeginBackEnd: function: %s, loop:%d---\r\n", this->GetJnFunction()->GetDisplayName(),

static\_cast<JsLoopBodyCodeGen \*>(this->m\_workItem)->GetLoopNumber());

}

else

{

Output::Print(L"---BeginBackEnd: function: %s---\r\n", this->GetJnFunction()->GetDisplayName());

}

Output::Flush();

}

#endif

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

LARGE\_INTEGER start\_time = { 0 };

if(PHASE\_TRACE(Js::BackEndPhase, GetJnFunction()))

{

QueryPerformanceCounter(&start\_time);

if (this->IsLoopBody())

{

Output::Print(

L"BeginBackEnd - function: %s (%s, line %u), loop: %u, mode: %S",

GetJnFunction()->GetDisplayName(),

GetJnFunction()->GetDebugNumberSet(debugStringBuffer),

GetJnFunction()->GetLineNumber(),

static\_cast<JsLoopBodyCodeGen \*>(this->m\_workItem)->GetLoopNumber(),

ExecutionModeName(m\_workItem->GetJitMode()));

if (this->m\_jnFunction->GetIsAsmjsMode())

{

Output::Print(L" (Asmjs)\n");

}

else

{

Output::Print(L"\n");

}

}

else

{

Output::Print(

L"BeginBackEnd - function: %s (%s, line %u), mode: %S",

GetJnFunction()->GetDisplayName(),

GetJnFunction()->GetDebugNumberSet(debugStringBuffer),

GetJnFunction()->GetLineNumber(),

ExecutionModeName(m\_workItem->GetJitMode()));

if (this->m\_jnFunction->GetIsAsmjsMode())

{

Output::Print(L" (Asmjs)\n");

}

else

{

Output::Print(L"\n");

}

}

Output::Flush();

}

#ifdef FIELD\_ACCESS\_STATS

if (PHASE\_TRACE(Js::ObjTypeSpecPhase, this->GetJnFunction()) || PHASE\_TRACE(Js::EquivObjTypeSpecPhase, this->GetJnFunction()))

{

if (this->m\_jitTimeData->inlineCacheStats)

{

auto stats = this->m\_jitTimeData->inlineCacheStats;

Output::Print(L"ObjTypeSpec: jitting function %s (#%s): inline cache stats:\n", this->GetJnFunction()->GetDisplayName(), this->GetJnFunction()->GetDebugNumberSet(debugStringBuffer));

Output::Print(L" overall: total %u, no profile info %u\n", stats->totalInlineCacheCount, stats->noInfoInlineCacheCount);

Output::Print(L" mono: total %u, empty %u, cloned %u\n",

stats->monoInlineCacheCount, stats->emptyMonoInlineCacheCount, stats->clonedMonoInlineCacheCount);

Output::Print(L" poly: total %u (high %u, low %u), null %u, empty %u, ignored %u, disabled %u, equivalent %u, non-equivalent %u, cloned %u\n",

stats->polyInlineCacheCount, stats->highUtilPolyInlineCacheCount, stats->lowUtilPolyInlineCacheCount,

stats->nullPolyInlineCacheCount, stats->emptyPolyInlineCacheCount, stats->ignoredPolyInlineCacheCount, stats->disabledPolyInlineCacheCount,

stats->equivPolyInlineCacheCount, stats->nonEquivPolyInlineCacheCount, stats->clonedPolyInlineCacheCount);

}

else

{

Output::Print(L"EquivObjTypeSpec: function %s (%s): inline cache stats unavailable\n", this->GetJnFunction()->GetDisplayName(), this->GetJnFunction()->GetDebugNumberSet(debugStringBuffer));

}

Output::Flush();

}

#endif

BEGIN\_CODEGEN\_PHASE(this, Js::BackEndPhase);

{

// IRBuilder

BEGIN\_CODEGEN\_PHASE(this, Js::IRBuilderPhase);

if (m\_jnFunction->GetIsAsmjsMode())

{

IRBuilderAsmJs asmIrBuilder(this);

asmIrBuilder.Build();

}

else

{

IRBuilder irBuilder(this);

irBuilder.Build();

}

END\_CODEGEN\_PHASE(this, Js::IRBuilderPhase);

#ifdef IR\_VIEWER

IRtoJSObjectBuilder::DumpIRtoGlobalObject(this, Js::IRBuilderPhase);

#endif /\* IR\_VIEWER \*/

BEGIN\_CODEGEN\_PHASE(this, Js::InlinePhase);

InliningHeuristics heuristics(this->GetJnFunction());

Inline inliner(this, heuristics);

inliner.Optimize();

END\_CODEGEN\_PHASE(this, Js::InlinePhase);

if (scriptContext->IsClosed())

{

// Should not be jitting something in the foreground when the script context is actually closed

Assert(IsBackgroundJIT() || !scriptContext->IsActuallyClosed());

throw Js::OperationAbortedException();

}

// FlowGraph

{

// Scope for FlowGraph arena

NoRecoverMemoryJitArenaAllocator fgAlloc(L"BE-FlowGraph", m\_alloc->GetPageAllocator(), Js::Throw::OutOfMemory);

BEGIN\_CODEGEN\_PHASE(this, Js::FGBuildPhase);

this->m\_fg = FlowGraph::New(this, &fgAlloc);

this->m\_fg->Build();

END\_CODEGEN\_PHASE(this, Js::FGBuildPhase);

// Global Optimization and Type Specialization

BEGIN\_CODEGEN\_PHASE(this, Js::GlobOptPhase);

GlobOpt globOpt(this);

globOpt.Optimize();

END\_CODEGEN\_PHASE(this, Js::GlobOptPhase);

// Delete flowGraph now

this->m\_fg->Destroy();

this->m\_fg = nullptr;

}

#ifdef IR\_VIEWER

IRtoJSObjectBuilder::DumpIRtoGlobalObject(this, Js::GlobOptPhase);

#endif /\* IR\_VIEWER \*/

ThrowIfScriptClosed();

// Lowering

Lowerer lowerer(this);

BEGIN\_CODEGEN\_PHASE(this, Js::LowererPhase);

lowerer.Lower();

END\_CODEGEN\_PHASE(this, Js::LowererPhase);

#ifdef IR\_VIEWER

IRtoJSObjectBuilder::DumpIRtoGlobalObject(this, Js::LowererPhase);

#endif /\* IR\_VIEWER \*/

// Encode constants

Security security(this);

BEGIN\_CODEGEN\_PHASE(this, Js::EncodeConstantsPhase)

security.EncodeLargeConstants();

END\_CODEGEN\_PHASE(this, Js::EncodeConstantsPhase);

if (this->GetScriptContext()->GetThreadContext()->DoInterruptProbe(this->GetJnFunction()))

{

BEGIN\_CODEGEN\_PHASE(this, Js::InterruptProbePhase)

lowerer.DoInterruptProbes();

END\_CODEGEN\_PHASE(this, Js::InterruptProbePhase)

}

// Register Allocation

BEGIN\_CODEGEN\_PHASE(this, Js::RegAllocPhase);

LinearScan linearScan(this);

linearScan.RegAlloc();

END\_CODEGEN\_PHASE(this, Js::RegAllocPhase);

#ifdef IR\_VIEWER

IRtoJSObjectBuilder::DumpIRtoGlobalObject(this, Js::RegAllocPhase);

#endif /\* IR\_VIEWER \*/

ThrowIfScriptClosed();

// Peephole optimizations

BEGIN\_CODEGEN\_PHASE(this, Js::PeepsPhase);

Peeps peeps(this);

peeps.PeepFunc();

END\_CODEGEN\_PHASE(this, Js::PeepsPhase);

// Layout

BEGIN\_CODEGEN\_PHASE(this, Js::LayoutPhase);

SimpleLayout layout(this);

layout.Layout();

END\_CODEGEN\_PHASE(this, Js::LayoutPhase);

if (this->HasTry() && this->hasBailoutInEHRegion)

{

BEGIN\_CODEGEN\_PHASE(this, Js::EHBailoutPatchUpPhase);

lowerer.EHBailoutPatchUp();

END\_CODEGEN\_PHASE(this, Js::EHBailoutPatchUpPhase);

}

// Insert NOPs (moving this before prolog/epilog for AMD64 and possibly ARM).

BEGIN\_CODEGEN\_PHASE(this, Js::InsertNOPsPhase);

security.InsertNOPs();

END\_CODEGEN\_PHASE(this, Js::InsertNOPsPhase);

// Prolog/Epilog

BEGIN\_CODEGEN\_PHASE(this, Js::PrologEpilogPhase);

if (m\_jnFunction->GetIsAsmjsMode())

{

lowerer.LowerPrologEpilogAsmJs();

}

else

{

lowerer.LowerPrologEpilog();

}

END\_CODEGEN\_PHASE(this, Js::PrologEpilogPhase);

BEGIN\_CODEGEN\_PHASE(this, Js::FinalLowerPhase);

lowerer.FinalLower();

END\_CODEGEN\_PHASE(this, Js::FinalLowerPhase);

// Encoder

BEGIN\_CODEGEN\_PHASE(this, Js::EncoderPhase);

Encoder encoder(this);

encoder.Encode();

END\_CODEGEN\_PHASE\_NO\_DUMP(this, Js::EncoderPhase);

#ifdef IR\_VIEWER

IRtoJSObjectBuilder::DumpIRtoGlobalObject(this, Js::EncoderPhase);

#endif /\* IR\_VIEWER \*/

}

END\_CODEGEN\_PHASE(this, Js::BackEndPhase);

#if DBG\_DUMP

if (Js::Configuration::Global.flags.TestTrace.IsEnabled(Js::BackEndPhase))

{

Output::Print(L"---EndBackEnd---\r\n");

Output::Flush();

}

#endif

#ifdef PROFILE\_BAILOUT\_RECORD\_MEMORY

if (Js::Configuration::Global.flags.ProfileBailOutRecordMemory)

{

scriptContext->codeSize += this->m\_codeSize;

}

#endif

if (PHASE\_TRACE(Js::BackEndPhase, GetJnFunction()))

{

LARGE\_INTEGER freq;

LARGE\_INTEGER end\_time;

QueryPerformanceCounter(&end\_time);

QueryPerformanceFrequency(&freq);

if (this->IsLoopBody())

{

Output::Print(

L"EndBackEnd - function: %s (%s, line %u), loop: %u, mode: %S, time:%8.6f mSec",

GetJnFunction()->GetDisplayName(),

GetJnFunction()->GetDebugNumberSet(debugStringBuffer),

GetJnFunction()->GetLineNumber(),

static\_cast<JsLoopBodyCodeGen \*>(this->m\_workItem)->GetLoopNumber(),

ExecutionModeName(m\_workItem->GetJitMode()),

(((double)((end\_time.QuadPart - start\_time.QuadPart)\* (double)1000.0 / (double)freq.QuadPart))) / (1));

if (this->m\_jnFunction->GetIsAsmjsMode())

{

Output::Print(L" (Asmjs)\n");

}

else

{

Output::Print(L"\n");

}

}

else

{

Output::Print(

L"EndBackEnd - function: %s (%s, line %u), mode: %S time:%8.6f mSec",

GetJnFunction()->GetDisplayName(),

GetJnFunction()->GetDebugNumberSet(debugStringBuffer),

GetJnFunction()->GetLineNumber(),

ExecutionModeName(m\_workItem->GetJitMode()),

(((double)((end\_time.QuadPart - start\_time.QuadPart)\* (double)1000.0 / (double)freq.QuadPart))) / (1));

if (this->m\_jnFunction->GetIsAsmjsMode())

{

Output::Print(L" (Asmjs)\n");

}

else

{

Output::Print(L"\n");

}

}

Output::Flush();

}

{

if(IS\_JS\_ETW(EventEnabledJSCRIPT\_FUNCTION\_JIT\_STOP()))

{

WCHAR displayNameBuffer[256];

WCHAR\* displayName = displayNameBuffer;

size\_t sizeInChars = this->m\_workItem->GetDisplayName(displayName, 256);

if(sizeInChars > 256)

{

displayName = new WCHAR[sizeInChars];

this->m\_workItem->GetDisplayName(displayName, 256);

}

void\* entryPoint;

ptrdiff\_t codeSize;

this->m\_workItem->GetEntryPointAddress(&entryPoint, &codeSize);

JS\_ETW(EventWriteJSCRIPT\_FUNCTION\_JIT\_STOP(

this->GetFunctionNumber(),

displayName,

scriptContext,

this->m\_workItem->GetInterpretedCount(),

entryPoint,

codeSize));

if(displayName != displayNameBuffer)

{

delete[] displayName;

}

}

}

#if DBG\_DUMP

if (Js::Configuration::Global.flags.IsEnabled(Js::AsmDumpModeFlag))

{

FILE \* oldFile = 0;

FILE \* asmFile = scriptContext->GetNativeCodeGenerator()->asmFile;

if (asmFile)

{

oldFile = Output::SetFile(asmFile);

}

this->Dump(IRDumpFlags\_AsmDumpMode);

Output::Flush();

if (asmFile)

{

FILE \*openedFile = Output::SetFile(oldFile);

Assert(openedFile == asmFile);

}

}

#endif

}

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

/// Func::StackAllocate

/// Allocate stack space of given size.

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

int32

Func::StackAllocate(int size)

{

Assert(this->IsTopFunc());

int32 offset;

#ifdef MD\_GROW\_LOCALS\_AREA\_UP

// Locals have positive offsets and are allocated from bottom to top.

m\_localStackHeight = Math::Align(m\_localStackHeight, min(size, MachStackAlignment));

offset = m\_localStackHeight;

m\_localStackHeight += size;

#else

// Locals have negative offsets and are allocated from top to bottom.

m\_localStackHeight += size;

m\_localStackHeight = Math::Align(m\_localStackHeight, min(size, MachStackAlignment));

offset = -m\_localStackHeight;

#endif

return offset;

}

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

///

/// Func::StackAllocate

///

/// Allocate stack space for this symbol.

///

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

int32

Func::StackAllocate(StackSym \*stackSym, int size)

{

Assert(size > 0);

if (stackSym->IsArgSlotSym() || stackSym->IsParamSlotSym() || stackSym->IsAllocated())

{

return stackSym->m\_offset;

}

Assert(stackSym->m\_offset == 0);

stackSym->m\_allocated = true;

stackSym->m\_offset = StackAllocate(size);

return stackSym->m\_offset;

}

void

Func::SetArgOffset(StackSym \*stackSym, int32 offset)

{

AssertMsg(offset >= 0, "Why is the offset, negative?");

stackSym->m\_offset = offset;

stackSym->m\_allocated = true;

}

///

/// Ensures that local var slots are created, if the function has locals.

/// Allocate stack space for locals used for debugging

/// (for local non-temp vars we write-through memory so that locals inspection can make use of that.).

// On stack, after local slots we allocate space for metadata (in particular, whether any the locals was changed in debugger).

///

void

Func::EnsureLocalVarSlots()

{

Assert(IsJitInDebugMode());

if (!this->HasLocalVarSlotCreated())

{

Assert(this->m\_jnFunction != nullptr);

uint32 localSlotCount = this->m\_jnFunction->GetNonTempLocalVarCount();

if (localSlotCount && m\_localVarSlotsOffset == Js::Constants::InvalidOffset)

{

// Allocate the slots.

int32 size = localSlotCount \* GetDiagLocalSlotSize();

m\_localVarSlotsOffset = StackAllocate(size);

m\_hasLocalVarChangedOffset = StackAllocate(max(1, MachStackAlignment)); // Can't alloc less than StackAlignment bytes.

Assert(this->m\_workItem->Type() == JsFunctionType);

// Store in the entry point info, so that it will later be used when we do the variable inspection.

Js::FunctionEntryPointInfo \* entryPointInfo = static\_cast<Js::FunctionEntryPointInfo\*>(this->m\_workItem->GetEntryPoint());

Assert(entryPointInfo != nullptr);

entryPointInfo->localVarSlotsOffset = AdjustOffsetValue(m\_localVarSlotsOffset);

entryPointInfo->localVarChangedOffset = AdjustOffsetValue(m\_hasLocalVarChangedOffset);

}

}

}

void Func::SetFirstArgOffset(IR::Instr\* inlineeStart)

{

Assert(inlineeStart->m\_func == this);

Assert(!IsTopFunc());

int32 lastOffset;

IR::Instr\* arg = inlineeStart->GetNextArg();

const auto lastArgOutStackSym = arg->GetDst()->AsSymOpnd()->m\_sym->AsStackSym();

lastOffset = lastArgOutStackSym->m\_offset;

Assert(lastArgOutStackSym->m\_isSingleDef);

const auto secondLastArgOutOpnd = lastArgOutStackSym->m\_instrDef->GetSrc2();

if (secondLastArgOutOpnd->IsSymOpnd())

{

const auto secondLastOffset = secondLastArgOutOpnd->AsSymOpnd()->m\_sym->AsStackSym()->m\_offset;

if (secondLastOffset > lastOffset)

{

lastOffset = secondLastOffset;

}

}

lastOffset += MachPtr;

int32 firstActualStackOffset = lastOffset - ((this->actualCount + Js::Constants::InlineeMetaArgCount) \* MachPtr);

Assert((this->firstActualStackOffset == -1) || (this->firstActualStackOffset == firstActualStackOffset));

this->firstActualStackOffset = firstActualStackOffset;

}

int32

Func::GetLocalVarSlotOffset(int32 slotId)

{

this->EnsureLocalVarSlots();

Assert(m\_localVarSlotsOffset != Js::Constants::InvalidOffset);

int32 slotOffset = slotId \* GetDiagLocalSlotSize();

return m\_localVarSlotsOffset + slotOffset;

}

void Func::OnAddSym(Sym\* sym)

{

Assert(sym);

if (this->IsJitInDebugMode() && this->IsNonTempLocalVar(sym->m\_id))

{

Assert(m\_nonTempLocalVars);

m\_nonTempLocalVars->Set(sym->m\_id);

}

}

///

/// Returns offset of the flag (1 byte) whether any local was changed (in debugger).

/// If the function does not have any locals, returns -1.

///

int32

Func::GetHasLocalVarChangedOffset()

{

this->EnsureLocalVarSlots();

return m\_hasLocalVarChangedOffset;

}

bool

Func::IsJitInDebugMode()

{

return

Js::Configuration::Global.EnableJitInDebugMode() &&

this->m\_workItem->IsJitInDebugMode();

}

bool

Func::IsNonTempLocalVar(uint32 slotIndex)

{

Assert(this->m\_jnFunction != nullptr);

return this->m\_jnFunction->IsNonTempLocalVar(slotIndex);

}

int32

Func::AdjustOffsetValue(int32 offset)

{

#ifdef MD\_GROW\_LOCALS\_AREA\_UP

return -(offset + BailOutInfo::StackSymBias);

#else

// Stack offset are negative, includes the PUSH EBP and return address

return offset - (2 \* MachPtr);

#endif

}

#ifdef MD\_GROW\_LOCALS\_AREA\_UP

// Note: this is called during jit-compile when we finalize bail out record.

void

Func::AjustLocalVarSlotOffset()

{

if (m\_jnFunction->GetNonTempLocalVarCount())

{

// Turn positive SP-relative base locals offset into negative frame-pointer-relative offset

// This is changing value for restoring the locals when read due to locals inspection.

int localsOffset = m\_localVarSlotsOffset - (m\_localStackHeight + m\_ArgumentsOffset);

int valueChangeOffset = m\_hasLocalVarChangedOffset - (m\_localStackHeight + m\_ArgumentsOffset);

Js::FunctionEntryPointInfo \* entryPointInfo = static\_cast<Js::FunctionEntryPointInfo\*>(this->m\_workItem->GetEntryPoint());

Assert(entryPointInfo != nullptr);

entryPointInfo->localVarSlotsOffset = localsOffset;

entryPointInfo->localVarChangedOffset = valueChangeOffset;

}

}

#endif

bool

Func::DoGlobOptsForGeneratorFunc()

{

// Disable GlobOpt optimizations for generators initially. Will visit and enable each one by one.

return !m\_jnFunction->IsGenerator();

}

void

Func::SetDoFastPaths()

{

// Make sure we only call this once!

Assert(!this->hasCalledSetDoFastPaths);

bool isLeaf = this->m\_isLeaf && !PHASE\_OFF(Js::LeafFastPathPhase, this);

bool doFastPaths = false;

if(!PHASE\_OFF(Js::FastPathPhase, this) && (!IsSimpleJit() || Js::FunctionBody::IsNewSimpleJit()))

{

if (isLeaf || this->GetScriptContext()->GetThreadContext()->GetSourceSize() < (size\_t)CONFIG\_FLAG(FastPathCap) || CONFIG\_FLAG(ForceFastPath))

{

doFastPaths = true;

}

}

this->m\_doFastPaths = doFastPaths;

#ifdef DBG

this->hasCalledSetDoFastPaths = true;

#endif

}

#ifdef \_M\_ARM

RegNum

Func::GetLocalsPointer() const

{

#ifdef DBG

if (Js::Configuration::Global.flags.IsEnabled(Js::ForceLocalsPtrFlag))

{

return ALT\_LOCALS\_PTR;

}

#endif

if (this->m\_jnFunction->GetHasTry())

{

return ALT\_LOCALS\_PTR;

}

return RegSP;

}

#endif

void Func::AddSlotArrayCheck(IR::SymOpnd \*fieldOpnd)

{

if (PHASE\_OFF(Js::ClosureRangeCheckPhase, this))

{

return;

}

Assert(IsTopFunc());

if (this->slotArrayCheckTable == nullptr)

{

this->slotArrayCheckTable = SlotArrayCheckTable::New(m\_alloc, 4);

}

PropertySym \*propertySym = fieldOpnd->m\_sym->AsPropertySym();

uint32 slot = propertySym->m\_propertyId;

uint32 \*pSlotId = this->slotArrayCheckTable->FindOrInsert(slot, propertySym->m\_stackSym->m\_id);

if (pSlotId && (\*pSlotId == (uint32)-1 || \*pSlotId < slot))

{

\*pSlotId = propertySym->m\_propertyId;

}

}

void Func::AddFrameDisplayCheck(IR::SymOpnd \*fieldOpnd, uint32 slotId)

{

if (PHASE\_OFF(Js::ClosureRangeCheckPhase, this))

{

return;

}

Assert(IsTopFunc());

if (this->frameDisplayCheckTable == nullptr)

{

this->frameDisplayCheckTable = FrameDisplayCheckTable::New(m\_alloc, 4);

}

PropertySym \*propertySym = fieldOpnd->m\_sym->AsPropertySym();

FrameDisplayCheckRecord \*\*record = this->frameDisplayCheckTable->FindOrInsertNew(propertySym->m\_stackSym->m\_id);

if (\*record == nullptr)

{

\*record = JitAnew(m\_alloc, FrameDisplayCheckRecord);

}

uint32 frameDisplaySlot = propertySym->m\_propertyId;

if ((\*record)->table == nullptr || (\*record)->slotId < frameDisplaySlot)

{

(\*record)->slotId = frameDisplaySlot;

}

if (slotId != (uint32)-1)

{

if ((\*record)->table == nullptr)

{

(\*record)->table = SlotArrayCheckTable::New(m\_alloc, 4);

}

uint32 \*pSlotId = (\*record)->table->FindOrInsert(slotId, frameDisplaySlot);

if (pSlotId && \*pSlotId < slotId)

{

\*pSlotId = slotId;

}

}

}

void Func::InitLocalClosureSyms()

{

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

// Allocate stack space for closure pointers. Do this only if we're jitting for stack closures, and

// tell bailout that these are not byte code symbols so that we don't try to encode them in the bailout record,

// as they don't have normal lifetimes.

Js::RegSlot regSlot = this->GetJnFunction()->GetLocalClosureReg();

if (regSlot != Js::Constants::NoRegister)

{

this->m\_localClosureSym =

StackSym::FindOrCreate(static\_cast<SymID>(regSlot),

this->DoStackFrameDisplay() ? (Js::RegSlot)-1 : regSlot,

this);

}

regSlot = this->GetJnFunction()->GetLocalFrameDisplayReg();

if (regSlot != Js::Constants::NoRegister)

{

this->m\_localFrameDisplaySym =

StackSym::FindOrCreate(static\_cast<SymID>(regSlot),

this->DoStackFrameDisplay() ? (Js::RegSlot)-1 : regSlot,

this);

}

}

bool Func::CanAllocInPreReservedHeapPageSegment ()

{

#ifdef \_CONTROL\_FLOW\_GUARD

return PHASE\_FORCE1(Js::PreReservedHeapAllocPhase) || (!PHASE\_OFF1(Js::PreReservedHeapAllocPhase) &&

!IsJitInDebugMode() && !GetScriptContext()->IsInDebugMode() && GetScriptContext()->GetThreadContext()->IsCFGEnabled()

#if \_M\_IX86

&& m\_workItem->GetJitMode() == ExecutionMode::FullJit && GetCodeGenAllocators()->canCreatePreReservedSegment);

#elif \_M\_X64

&& true);

#else

&& false); //Not yet implemented for architectures other than x86 and amd64.

#endif //\_M\_ARCH

#else

return false;

#endif//\_CONTROL\_FLOW\_GUARD

}

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

///

/// Func::GetInstrCount

///

/// Returns the number of instrs.

/// Note: It counts all instrs for now, including labels, etc.

///

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

uint32

Func::GetInstrCount()

{

uint instrCount = 0;

FOREACH\_INSTR\_IN\_FUNC(instr, this)

{

instrCount++;

}NEXT\_INSTR\_IN\_FUNC;

return instrCount;

}

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

///

/// Func::NumberInstrs

///

/// Number each instruction in order of appearance in the function.

///

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

void

Func::NumberInstrs()

{

#if DBG\_DUMP

Assert(this->IsTopFunc());

Assert(!this->hasInstrNumber);

this->hasInstrNumber = true;

#endif

uint instrCount = 1;

FOREACH\_INSTR\_IN\_FUNC(instr, this)

{

instr->SetNumber(instrCount++);

}

NEXT\_INSTR\_IN\_FUNC;

}

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

///

/// Func::IsInPhase

///

/// Determines whether the function is currently in the provided phase

///

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

#if DBG

bool

Func::IsInPhase(Js::Phase tag)

{

return this->GetTopFunc()->currentPhases.Contains(tag);

}

#endif

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

///

/// Func::BeginPhase

///

/// Takes care of the profiler

///

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

void

Func::BeginPhase(Js::Phase tag)

{

#ifdef DBG

this->GetTopFunc()->currentPhases.Push(tag);

#endif

#ifdef PROFILE\_EXEC

AssertMsg((this->m\_codeGenProfiler != nullptr) == Js::Configuration::Global.flags.IsEnabled(Js::ProfileFlag),

"Profiler tag is supplied but the profiler pointer is NULL");

if (this->m\_codeGenProfiler)

{

this->m\_codeGenProfiler->ProfileBegin(tag);

}

#endif

}

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

///

/// Func::EndPhase

///

/// Takes care of the profiler and dumper

///

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

void

Func::EndProfiler(Js::Phase tag)

{

#ifdef DBG

Assert(this->GetTopFunc()->currentPhases.Count() > 0);

Js::Phase popped = this->GetTopFunc()->currentPhases.Pop();

Assert(tag == popped);

#endif

#ifdef PROFILE\_EXEC

AssertMsg((this->m\_codeGenProfiler != nullptr) == Js::Configuration::Global.flags.IsEnabled(Js::ProfileFlag),

"Profiler tag is supplied but the profiler pointer is NULL");

if (this->m\_codeGenProfiler)

{

this->m\_codeGenProfiler->ProfileEnd(tag);

}

#endif

}

void

Func::EndPhase(Js::Phase tag, bool dump)

{

this->EndProfiler(tag);

#if DBG\_DUMP

if(dump && (PHASE\_DUMP(tag, this)

|| PHASE\_DUMP(Js::BackEndPhase, this)))

{

Output::Print(L"-----------------------------------------------------------------------------\n");

if (m\_workItem->Type() == JsLoopBodyWorkItemType)

{

Output::Print(L"\*\*\*\*\*\*\*\*\*\*\*\* IR after %s (%S) Loop %d \*\*\*\*\*\*\*\*\*\*\*\*\n", Js::PhaseNames[tag], ExecutionModeName(m\_workItem->GetJitMode()), ((JsLoopBodyCodeGen\*)m\_workItem)->GetLoopNumber());

}

else

{

Output::Print(L"\*\*\*\*\*\*\*\*\*\*\*\* IR after %s (%S) \*\*\*\*\*\*\*\*\*\*\*\*\n", Js::PhaseNames[tag], ExecutionModeName(m\_workItem->GetJitMode()));

}

this->Dump(Js::Configuration::Global.flags.AsmDiff? IRDumpFlags\_AsmDumpMode : IRDumpFlags\_None);

}

#endif

#if DBG

if (tag == Js::LowererPhase)

{

Assert(!this->isPostLower);

this->isPostLower = true;

}

else if (tag == Js::RegAllocPhase)

{

Assert(!this->isPostRegAlloc);

this->isPostRegAlloc = true;

}

else if (tag == Js::PeepsPhase)

{

Assert(this->isPostLower && !this->isPostLayout);

this->isPostPeeps = true;

}

else if (tag == Js::LayoutPhase)

{

Assert(this->isPostPeeps && !this->isPostLayout);

this->isPostLayout = true;

}

else if (tag == Js::FinalLowerPhase)

{

Assert(this->isPostLayout && !this->isPostFinalLower);

this->isPostFinalLower = true;

}

if (this->isPostLower)

{

#ifndef \_M\_ARM // Need to verify ARM is clean.

DbCheckPostLower dbCheck(this);

dbCheck.Check();

#endif

}

#endif

}

Func const \*

Func::GetTopFunc() const

{

Func const \* func = this;

while (!func->IsTopFunc())

{

func = func->parentFunc;

}

return func;

}

Func \*

Func::GetTopFunc()

{

Func \* func = this;

while (!func->IsTopFunc())

{

func = func->parentFunc;

}

return func;

}

StackSym \*

Func::EnsureLoopParamSym()

{

if (this->m\_loopParamSym == nullptr)

{

this->m\_loopParamSym = StackSym::New(TyMachPtr, this);

}

return this->m\_loopParamSym;

}

void

Func::UpdateMaxInlineeArgOutCount(uint inlineeArgOutCount)

{

if (maxInlineeArgOutCount < inlineeArgOutCount)

{

maxInlineeArgOutCount = inlineeArgOutCount;

}

}

void

Func::BeginClone(Lowerer \* lowerer, JitArenaAllocator \*alloc)

{

Assert(this->IsTopFunc());

AssertMsg(m\_cloner == nullptr, "Starting new clone while one is in progress");

m\_cloner = JitAnew(alloc, Cloner, lowerer, alloc);

if (m\_cloneMap == nullptr)

{

m\_cloneMap = JitAnew(alloc, InstrMap, alloc, 7);

}

}

void

Func::EndClone()

{

Assert(this->IsTopFunc());

if (m\_cloner)

{

m\_cloner->Finish();

JitAdelete(m\_cloner->alloc, m\_cloner);

m\_cloner = nullptr;

}

}

IR::SymOpnd \*

Func::GetInlineeOpndAtOffset(int32 offset)

{

Assert(IsInlinee());

StackSym \*stackSym = CreateInlineeStackSym();

this->SetArgOffset(stackSym, stackSym->m\_offset + offset);

Assert(stackSym->m\_offset >= 0);

return IR::SymOpnd::New(stackSym, 0, TyMachReg, this);

}

StackSym \*

Func::CreateInlineeStackSym()

{

// Make sure this is an inlinee and that GlobOpt has initialized the offset

// in the inlinee's frame.

Assert(IsInlinee());

Assert(m\_inlineeFrameStartSym->m\_offset != -1);

StackSym \*stackSym = m\_symTable->GetArgSlotSym((Js::ArgSlot)-1);

stackSym->m\_isInlinedArgSlot = true;

stackSym->m\_offset = m\_inlineeFrameStartSym->m\_offset;

stackSym->m\_allocated = true;

return stackSym;

}

uint8 \*

Func::GetCallsCountAddress() const

{

Assert(this->m\_workItem->Type() == JsFunctionType);

JsFunctionCodeGen \* functionCodeGen = static\_cast<JsFunctionCodeGen \*>(this->m\_workItem);

return functionCodeGen->GetFunctionBody()->GetCallsCountAddress(functionCodeGen->GetEntryPoint());

}

RecyclerWeakReference<Js::FunctionBody> \*

Func::GetWeakFuncRef() const

{

if (this->m\_jitTimeData == nullptr)

{

return nullptr;

}

return this->m\_jitTimeData->GetWeakFuncRef();

}

Js::InlineCache \*

Func::GetRuntimeInlineCache(const uint index) const

{

if(this->m\_runtimeData)

{

const auto inlineCache = this->m\_runtimeData->ClonedInlineCaches()->GetInlineCache(this->m\_jnFunction, index);

if(inlineCache)

{

return inlineCache;

}

}

return this->m\_jnFunction->GetInlineCache(index);

}

Js::PolymorphicInlineCache \*

Func::GetRuntimePolymorphicInlineCache(const uint index) const

{

if (this->m\_polymorphicInlineCacheInfo)

{

return this->m\_polymorphicInlineCacheInfo->GetPolymorphicInlineCaches()->GetInlineCache(this->m\_jnFunction, index);

}

return nullptr;

}

byte

Func::GetPolyCacheUtilToInitialize(const uint index) const

{

return this->GetRuntimePolymorphicInlineCache(index) ? this->GetPolyCacheUtil(index) : PolymorphicInlineCacheUtilizationMinValue;

}

byte

Func::GetPolyCacheUtil(const uint index) const

{

return this->m\_polymorphicInlineCacheInfo->GetUtilArray()->GetUtil(this->m\_jnFunction, index);

}

Js::ObjTypeSpecFldInfo\*

Func::GetObjTypeSpecFldInfo(const uint index) const

{

if (this->m\_jitTimeData == nullptr)

{

return nullptr;

}

Assert(this->m\_jitTimeData->GetObjTypeSpecFldInfoArray());

return this->m\_jitTimeData->GetObjTypeSpecFldInfoArray()->GetInfo(this->m\_jnFunction, index);

}

Js::ObjTypeSpecFldInfo\*

Func::GetGlobalObjTypeSpecFldInfo(uint propertyInfoId) const

{

Assert(this->m\_jitTimeData != nullptr);

return this->m\_jitTimeData->GetGlobalObjTypeSpecFldInfo(propertyInfoId);

}

void

Func::SetGlobalObjTypeSpecFldInfo(uint propertyInfoId, Js::ObjTypeSpecFldInfo\* info)

{

Assert(this->m\_jitTimeData != nullptr);

this->m\_jitTimeData->SetGlobalObjTypeSpecFldInfo(propertyInfoId, info);

}

void

Func::EnsurePinnedTypeRefs()

{

if (this->pinnedTypeRefs == nullptr)

{

this->pinnedTypeRefs = JitAnew(this->m\_alloc, TypeRefSet, this->m\_alloc);

}

}

void

Func::PinTypeRef(void\* typeRef)

{

EnsurePinnedTypeRefs();

this->pinnedTypeRefs->AddNew(typeRef);

}

void

Func::EnsureSingleTypeGuards()

{

if (this->singleTypeGuards == nullptr)

{

this->singleTypeGuards = JitAnew(this->m\_alloc, TypePropertyGuardDictionary, this->m\_alloc);

}

}

Js::JitTypePropertyGuard\*

Func::GetOrCreateSingleTypeGuard(Js::Type\* type)

{

EnsureSingleTypeGuards();

Js::JitTypePropertyGuard\* guard;

if (!this->singleTypeGuards->TryGetValue(type, &guard))

{

// Property guards are allocated by NativeCodeData::Allocator so that their lifetime extends as long as the EntryPointInfo is alive.

guard = NativeCodeDataNew(GetNativeCodeDataAllocator(), Js::JitTypePropertyGuard, type, this->indexedPropertyGuardCount++);

this->singleTypeGuards->Add(type, guard);

}

else

{

Assert(guard->GetType() == type);

}

return guard;

}

void

Func::EnsureEquivalentTypeGuards()

{

if (this->equivalentTypeGuards == nullptr)

{

this->equivalentTypeGuards = JitAnew(this->m\_alloc, EquivalentTypeGuardList, this->m\_alloc);

}

}

Js::JitEquivalentTypeGuard\*

Func::CreateEquivalentTypeGuard(Js::Type\* type, uint32 objTypeSpecFldId)

{

EnsureEquivalentTypeGuards();

Js::JitEquivalentTypeGuard\* guard = NativeCodeDataNew(GetNativeCodeDataAllocator(), Js::JitEquivalentTypeGuard, type, this->indexedPropertyGuardCount++, objTypeSpecFldId);

// If we want to hard code the address of the cache, we will need to go back to allocating it from the native code data allocator.

// We would then need to maintain consistency (double write) to both the recycler allocated cache and the one on the heap.

Js::EquivalentTypeCache\* cache = NativeCodeDataNewZ(GetTransferDataAllocator(), Js::EquivalentTypeCache);

guard->SetCache(cache);

// Give the cache a back-pointer to the guard so that the guard can be cleared at runtime if necessary.

cache->SetGuard(guard);

this->equivalentTypeGuards->Prepend(guard);

return guard;

}

void

Func::EnsurePropertyGuardsByPropertyId()

{

if (this->propertyGuardsByPropertyId == nullptr)

{

this->propertyGuardsByPropertyId = JitAnew(this->m\_alloc, PropertyGuardByPropertyIdMap, this->m\_alloc);

}

}

void

Func::EnsureCtorCachesByPropertyId()

{

if (this->ctorCachesByPropertyId == nullptr)

{

this->ctorCachesByPropertyId = JitAnew(this->m\_alloc, CtorCachesByPropertyIdMap, this->m\_alloc);

}

}

void

Func::LinkGuardToPropertyId(Js::PropertyId propertyId, Js::JitIndexedPropertyGuard\* guard)

{

Assert(guard != nullptr);

Assert(guard->GetValue() != NULL);

Assert(this->propertyGuardsByPropertyId != nullptr);

IndexedPropertyGuardSet\* set;

if (!this->propertyGuardsByPropertyId->TryGetValue(propertyId, &set))

{

set = JitAnew(this->m\_alloc, IndexedPropertyGuardSet, this->m\_alloc);

this->propertyGuardsByPropertyId->Add(propertyId, set);

}

set->Item(guard);

}

void

Func::LinkCtorCacheToPropertyId(Js::PropertyId propertyId, Js::JitTimeConstructorCache\* cache)

{

Assert(cache != nullptr);

Assert(this->ctorCachesByPropertyId != nullptr);

CtorCacheSet\* set;

if (!this->ctorCachesByPropertyId->TryGetValue(propertyId, &set))

{

set = JitAnew(this->m\_alloc, CtorCacheSet, this->m\_alloc);

this->ctorCachesByPropertyId->Add(propertyId, set);

}

set->Item(cache->runtimeCache);

}

Js::JitTimeConstructorCache\* Func::GetConstructorCache(const Js::ProfileId profiledCallSiteId)

{

Assert(GetJnFunction() != nullptr);

Assert(profiledCallSiteId < GetJnFunction()->GetProfiledCallSiteCount());

Assert(this->constructorCaches != nullptr);

return this->constructorCaches[profiledCallSiteId];

}

void Func::SetConstructorCache(const Js::ProfileId profiledCallSiteId, Js::JitTimeConstructorCache\* constructorCache)

{

const auto functionBody = this->GetJnFunction();

Assert(functionBody != nullptr);

Assert(profiledCallSiteId < functionBody->GetProfiledCallSiteCount());

Assert(constructorCache != nullptr);

Assert(this->constructorCaches != nullptr);

Assert(this->constructorCaches[profiledCallSiteId] == nullptr);

this->constructorCacheCount++;

this->constructorCaches[profiledCallSiteId] = constructorCache;

}

void Func::EnsurePropertiesWrittenTo()

{

if (this->propertiesWrittenTo == nullptr)

{

this->propertiesWrittenTo = JitAnew(this->m\_alloc, PropertyIdSet, this->m\_alloc);

}

}

void Func::EnsureCallSiteToArgumentsOffsetFixupMap()

{

if (this->callSiteToArgumentsOffsetFixupMap == nullptr)

{

this->callSiteToArgumentsOffsetFixupMap = JitAnew(this->m\_alloc, CallSiteToArgumentsOffsetFixupMap, this->m\_alloc);

}

}

IR::LabelInstr \*

Func::GetFuncStartLabel()

{

return m\_funcStartLabel;

}

IR::LabelInstr \*

Func::EnsureFuncStartLabel()

{

if(m\_funcStartLabel == nullptr)

{

m\_funcStartLabel = IR::LabelInstr::New( Js::OpCode::Label, this );

}

return m\_funcStartLabel;

}

IR::LabelInstr \*

Func::GetFuncEndLabel()

{

return m\_funcEndLabel;

}

IR::LabelInstr \*

Func::EnsureFuncEndLabel()

{

if(m\_funcEndLabel == nullptr)

{

m\_funcEndLabel = IR::LabelInstr::New( Js::OpCode::Label, this );

}

return m\_funcEndLabel;

}

void

Cloner::AddInstr(IR::Instr \* instrOrig, IR::Instr \* instrClone)

{

if (!this->instrFirst)

{

this->instrFirst = instrClone;

}

this->instrLast = instrClone;

}

void

Cloner::Finish()

{

this->RetargetClonedBranches();

if (this->lowerer)

{

lowerer->LowerRange(this->instrFirst, this->instrLast, false, false);

}

}

void

Cloner::RetargetClonedBranches()

{

if (!this->fRetargetClonedBranch)

{

return;

}

FOREACH\_INSTR\_IN\_RANGE(instr, this->instrFirst, this->instrLast)

{

if (instr->IsBranchInstr())

{

instr->AsBranchInstr()->RetargetClonedBranch();

}

}

NEXT\_INSTR\_IN\_RANGE;

}

void Func::ThrowIfScriptClosed()

{

Js::ScriptContext\* scriptContext = this->GetScriptContext();

if(scriptContext->IsClosed())

{

// Should not be jitting something in the foreground when the script context is actually closed

Assert(IsBackgroundJIT() || !scriptContext->IsActuallyClosed());

throw Js::OperationAbortedException();

}

}

IR::IndirOpnd \* Func::GetConstantAddressIndirOpnd(void \* address, IR::AddrOpndKind kind, IRType type, Js::OpCode loadOpCode)

{

Assert(this->GetTopFunc() == this);

if (!canHoistConstantAddressLoad)

{

// We can't hoist constant address load after lower, as we can't mark the sym as

// live on back edge

return nullptr;

}

int offset = 0;

IR::RegOpnd \*\* foundRegOpnd = this->constantAddressRegOpnd.Find([address, &offset](IR::RegOpnd \* regOpnd)

{

Assert(regOpnd->m\_sym->IsSingleDef());

void \* curr = regOpnd->m\_sym->m\_instrDef->GetSrc1()->AsAddrOpnd()->m\_address;

ptrdiff\_t diff = (intptr\_t)address - (intptr\_t)curr;

if (!Math::FitsInDWord(diff))

{

return false;

}

offset = (int)diff;

return true;

});

IR::RegOpnd \* addressRegOpnd;

if (foundRegOpnd != nullptr)

{

addressRegOpnd = \*foundRegOpnd;

}

else

{

Assert(offset == 0);

addressRegOpnd = IR::RegOpnd::New(TyMachPtr, this);

IR::Instr \*const newInstr =

IR::Instr::New(

loadOpCode,

addressRegOpnd,

IR::AddrOpnd::New(address, kind, this, true),

this);

this->constantAddressRegOpnd.Prepend(addressRegOpnd);

IR::Instr \* insertBeforeInstr = this->lastConstantAddressRegLoadInstr;

if (insertBeforeInstr == nullptr)

{

insertBeforeInstr = this->GetFunctionEntryInsertionPoint();

this->lastConstantAddressRegLoadInstr = newInstr;

}

insertBeforeInstr->InsertBefore(newInstr);

}

IR::IndirOpnd \* indirOpnd = IR::IndirOpnd::New(addressRegOpnd, offset, type, this, true);

#if DBG\_DUMP

indirOpnd->SetAddrKind(kind, address);

#endif

return indirOpnd;

}

void Func::MarkConstantAddressSyms(BVSparse<JitArenaAllocator> \* bv)

{

Assert(this->GetTopFunc() == this);

this->constantAddressRegOpnd.Iterate([bv](IR::RegOpnd \* regOpnd)

{

bv->Set(regOpnd->m\_sym->m\_id);

});

}

IR::Instr \*

Func::GetFunctionEntryInsertionPoint()

{

Assert(this->GetTopFunc() == this);

IR::Instr \* insertInsert = this->lastConstantAddressRegLoadInstr;

if (insertInsert != nullptr)

{

return insertInsert->m\_next;

}

insertInsert = this->m\_headInstr;

if (this->HasTry())

{

// Insert it inside the root region

insertInsert = insertInsert->m\_next;

Assert(insertInsert->IsLabelInstr() && insertInsert->AsLabelInstr()->GetRegion()->GetType() == RegionTypeRoot);

}

return insertInsert->m\_next;

}

#if DBG\_DUMP

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

///

/// Func::DumpHeader

///

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

void

Func::DumpHeader()

{

Output::Print(L"-----------------------------------------------------------------------------\n");

this->m\_jnFunction->DumpFullFunctionName();

Output::SkipToColumn(50);

Output::Print(L"Instr Count:%d", GetInstrCount());

if(m\_codeSize > 0)

{

Output::Print(L"\t\tSize:%d\n\n", m\_codeSize);

}

else

{

Output::Print(L"\n\n");

}

}

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

///

/// Func::Dump

///

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

void

Func::Dump(IRDumpFlags flags)

{

this->DumpHeader();

FOREACH\_INSTR\_IN\_FUNC(instr, this)

{

instr->DumpGlobOptInstrString();

instr->Dump(flags);

}NEXT\_INSTR\_IN\_FUNC;

Output::Flush();

}

void

Func::Dump()

{

this->Dump(IRDumpFlags\_None);

}

#endif

#if DBG\_DUMP || defined(ENABLE\_IR\_VIEWER)

LPCSTR

Func::GetVtableName(INT\_PTR address)

{

#if DBG

if (vtableMap == nullptr)

{

vtableMap = VirtualTableRegistry::CreateVtableHashMap(this->m\_alloc);

};

LPCSTR name = vtableMap->Lookup(address, nullptr);

if (name)

{

if (strncmp(name, "class ", \_countof("class ") - 1) == 0)

{

name += \_countof("class ") - 1;

}

}

return name;

#else

return "";

#endif

}

#endif

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

bool Func::DoRecordNativeMap() const

{

#if defined(VTUNE\_PROFILING)

if (EtwTrace::isJitProfilingActive)

{

return true;

}

#endif

#if DBG\_DUMP

return PHASE\_DUMP(Js::EncoderPhase, this) && Js::Configuration::Global.flags.Verbose;

#else

return false;

#endif

}

#endif