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

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

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

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

#include "RuntimeByteCodePch.h"

#ifndef TEMP\_DISABLE\_ASMJS

#if DBG\_DUMP

#include "Language\AsmJsModule.h"

#include "ByteCode\AsmJSByteCodeDumper.h"

namespace Js

{

void AsmJsByteCodeDumper::Dump(AsmJsFunc\* func, FunctionBody\* body)

{

ByteCodeReader reader;

reader.Create(body);

StatementReader statementReader;

statementReader.Create(body);

body->DumpFullFunctionName();

Output::Print(L" Asm.js (");

const ArgSlot argCount = func->GetArgCount();

for (ArgSlot i = 0; i < argCount; i++)

{

AsmJsType var = func->GetArgType(i);

if (i > 0)

{

Output::Print(L", ");

}

if (var.isDouble())

{

Output::Print(L"+In%hu", i);

}

else if (var.isFloat())

{

Output::Print(L"flt(In%hu)", i);

}

else if (var.isInt())

{

Output::Print(L"In%hu|0", i);

}

else if (var.isSIMDType())

{

switch (var.GetWhich())

{

case AsmJsType::Int32x4:

Output::Print(L"I4(In%hu)", i);

break;

case AsmJsType::Float32x4:

Output::Print(L"F4(In%hu)", i);

break;

case AsmJsType::Float64x2:

Output::Print(L"D2(In%hu)", i);

break;

}

}

else

{

Assert(UNREACHED);

}

}

Output::Print(L") ");

Output::Print(L"(size: %d [%d])\n", body->GetByteCodeCount(), body->GetByteCodeWithoutLDACount());

const auto& intRegisters = func->GetRegisterSpace<int>();

const auto& doubleRegisters = func->GetRegisterSpace<double>();

const auto& floatRegisters = func->GetRegisterSpace<float>();

Output::Print(

L" Integer : %u locals (%u temps from I%u)\n",

intRegisters.GetVarCount(),

intRegisters.GetTmpCount(),

intRegisters.GetFirstTmpRegister());

Output::Print(

L" Doubles : %u locals (%u temps from D%u)\n",

doubleRegisters.GetVarCount(),

doubleRegisters.GetTmpCount(),

doubleRegisters.GetFirstTmpRegister());

Output::Print(

L" Floats : %u locals (%u temps from F%u)\n",

floatRegisters.GetVarCount(),

floatRegisters.GetTmpCount(),

floatRegisters.GetFirstTmpRegister());

const auto& simdRegisters = func->GetRegisterSpace<AsmJsSIMDValue>();

Output::Print(

L" SIMDs : %u locals (%u temps from SIMD%u)\n",

simdRegisters.GetVarCount(),

simdRegisters.GetTmpCount(),

simdRegisters.GetFirstTmpRegister());

uint32 statementIndex = 0;

DumpConstants(func, body);

Output::Print(L" Implicit Arg Ins:\n ======== =====\n ");

int iArg = intRegisters.GetConstCount(), dArg = doubleRegisters.GetConstCount(), fArg = floatRegisters.GetConstCount();

int simdArg = simdRegisters.GetConstCount();

for (ArgSlot i = 0; i < argCount; i++)

{

const AsmJsType& var = func->GetArgType(i);

if (var.isDouble())

{

Output::Print(L" D%d In%d", dArg++, i);

}

else if (var.isFloat())

{

Output::Print(L" F%d In%d", fArg++, i);

}

else if (var.isInt())

{

Output::Print(L" I%d In%d", iArg++, i);

}

else if (var.isSIMDType())

{

Output::Print(L" SIMD%d In%d", simdArg++, i);

}

else

{

Assert(UNREACHED);

}

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

}

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

if (func->GetReturnType() == AsmJsRetType::Void)

{

Output::Print(L" 0000 %-20s R0\n", OpCodeUtilAsmJs::GetOpCodeName(OpCodeAsmJs::LdUndef));

}

while (true)

{

while (statementReader.AtStatementBoundary(&reader))

{

body->PrintStatementSourceLine(statementIndex);

statementIndex = statementReader.MoveNextStatementBoundary();

}

int byteOffset = reader.GetCurrentOffset();

LayoutSize layoutSize;

OpCodeAsmJs op = (OpCodeAsmJs)reader.ReadOp(layoutSize);

if (op == OpCodeAsmJs::EndOfBlock)

{

Assert(reader.GetCurrentOffset() == body->GetByteCode()->GetLength());

break;

}

Output::Print(L" %04x %2s", byteOffset, layoutSize == LargeLayout ? L"L-" : layoutSize == MediumLayout ? L"M-" : L"");

DumpOp(op, layoutSize, reader, body);

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

{

int layoutStart = byteOffset + 2; // Account for the prefix op

int endByteOffset = reader.GetCurrentOffset();

Output::SkipToColumn(70);

if (layoutSize == LargeLayout)

{

Output::Print(L"%02X ",

op > Js::OpCodeAsmJs::MaxByteSizedOpcodes ?

Js::OpCodeAsmJs::ExtendedLargeLayoutPrefix : Js::OpCodeAsmJs::LargeLayoutPrefix);

}

else if (layoutSize == MediumLayout)

{

Output::Print(L"%02X ",

op > Js::OpCodeAsmJs::MaxByteSizedOpcodes ?

Js::OpCodeAsmJs::ExtendedMediumLayoutPrefix : Js::OpCodeAsmJs::MediumLayoutPrefix);

}

else

{

Assert(layoutSize == SmallLayout);

if (op > Js::OpCodeAsmJs::MaxByteSizedOpcodes)

{

Output::Print(L"%02X ", Js::OpCodeAsmJs::ExtendedOpcodePrefix);

}

else

{

Output::Print(L" ");

layoutStart--; // don't have a prefix

}

}

Output::Print(L"%02x", (byte)op);

for (int i = layoutStart; i < endByteOffset; i++)

{

Output::Print(L" %02x", reader.GetRawByte(i));

}

}

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

}

if (statementReader.AtStatementBoundary(&reader))

{

body->PrintStatementSourceLine(statementIndex);

statementIndex = statementReader.MoveNextStatementBoundary();

}

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

Output::Flush();

}

void AsmJsByteCodeDumper::DumpConstants(AsmJsFunc\* func, FunctionBody\* body)

{

const auto& intRegisters = func->GetRegisterSpace<int>();

const auto& doubleRegisters = func->GetRegisterSpace<double>();

const auto& floatRegisters = func->GetRegisterSpace<float>();

int nbIntConst = intRegisters.GetConstCount();

int nbDoubleConst = doubleRegisters.GetConstCount();

int nbFloatConst = floatRegisters.GetConstCount();

int\* constTable = (int\*)((Var\*)body->GetConstTable() + (AsmJsFunctionMemory::RequiredVarConstants - 1));

if (nbIntConst > 0)

{

Output::Print(L" Constant Integer:\n ======== =======\n ");

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

{

Output::Print(L" I%d %d\n ", i, \*constTable);

++constTable;

}

}

float\* floatTable = (float\*)constTable;

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

if (nbFloatConst > 0)

{

// const int inc = sizeof( double ) / sizeof( void\* );

Output::Print(L" Constant Floats:\n ======== ======\n ");

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

{

Output::Print(L" F%d %.4f\n ", i, \*floatTable);

++floatTable;

++constTable;

}

}

double\* doubleTable = (double\*)constTable;

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

if (nbDoubleConst > 0)

{

// const int inc = sizeof( double ) / sizeof( void\* );

Output::Print(L" Constant Doubles:\n ======== ======\n ");

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

{

Output::Print(L" D%d %.4f\n ", i, \*doubleTable);

++doubleTable;

}

}

// SIMD reg space is un-typed.

// We print each register in its 3 possible types to ease debugging.

const auto& simdRegisters = func->GetRegisterSpace<AsmJsSIMDValue>();

int nbSimdConst = simdRegisters.GetConstCount();

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

if (nbSimdConst > 0)

{

AsmJsSIMDValue\* simdTable = (AsmJsSIMDValue\*)doubleTable;

Output::Print(L" Constant SIMD values:\n ======== ======\n ");

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

{

Output::Print(L"SIMD%d ", i);

Output::Print(L"\tI4(%d, %d, %d, %d),", simdTable->i32[SIMD\_X], simdTable->i32[SIMD\_Y], simdTable->i32[SIMD\_Z], simdTable->i32[SIMD\_W]);

Output::Print(L"\tF4(%.4f, %.4f, %.4f, %.4f),", simdTable->f32[SIMD\_X], simdTable->f32[SIMD\_Y], simdTable->f32[SIMD\_Z], simdTable->f32[SIMD\_W]);

Output::Print(L"\tD2(%.4f, %.4f)\n ", simdTable->f64[SIMD\_X], simdTable->f64[SIMD\_Y]);

++simdTable;

}

}

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

}

void AsmJsByteCodeDumper::DumpOp(OpCodeAsmJs op, LayoutSize layoutSize, ByteCodeReader& reader, FunctionBody\* dumpFunction)

{

Output::Print(L"%-20s", OpCodeUtilAsmJs::GetOpCodeName(op));

OpLayoutTypeAsmJs nType = OpCodeUtilAsmJs::GetOpCodeLayout(op);

switch (layoutSize \* OpLayoutTypeAsmJs::Count + nType)

{

#define LAYOUT\_TYPE(layout) \

case OpLayoutTypeAsmJs::layout: \

Assert(layoutSize == SmallLayout); \

Dump##layout(op, reader.layout(), dumpFunction, reader); \

break;

#define LAYOUT\_TYPE\_WMS(layout) \

case SmallLayout \* OpLayoutTypeAsmJs::Count + OpLayoutTypeAsmJs::layout: \

Dump##layout(op, reader.layout##\_Small(), dumpFunction, reader); \

break; \

case MediumLayout \* OpLayoutTypeAsmJs::Count + OpLayoutTypeAsmJs::layout: \

Dump##layout(op, reader.layout##\_Medium(), dumpFunction, reader); \

break; \

case LargeLayout \* OpLayoutTypeAsmJs::Count + OpLayoutTypeAsmJs::layout: \

Dump##layout(op, reader.layout##\_Large(), dumpFunction, reader); \

break;

#include "LayoutTypesAsmJs.h"

default:

AssertMsg(false, "Unknown OpLayout");

break;

}

}

void AsmJsByteCodeDumper::DumpIntReg(RegSlot reg)

{

Output::Print(L" I%d ", (int)reg);

}

void AsmJsByteCodeDumper::DumpDoubleReg(RegSlot reg)

{

Output::Print(L" D%d ", (int)reg);

}

void AsmJsByteCodeDumper::DumpFloatReg(RegSlot reg)

{

Output::Print(L" F%d ", (int)reg);

}

void AsmJsByteCodeDumper::DumpR8Float(float value)

{

Output::Print(L" float:%f ", value);

}

// Float32x4

void AsmJsByteCodeDumper::DumpFloat32x4Reg(RegSlot reg)

{

Output::Print(L"F4\_%d ", (int)reg);

}

// Int32x4

void AsmJsByteCodeDumper::DumpInt32x4Reg(RegSlot reg)

{

Output::Print(L"I4\_%d ", (int)reg);

}

// Float64x2

void AsmJsByteCodeDumper::DumpFloat64x2Reg(RegSlot reg)

{

Output::Print(L"D2\_%d ", (int)reg);

}

template <class T>

void AsmJsByteCodeDumper::DumpElementSlot(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

switch (op)

{

case OpCodeAsmJs::LdSlot:

case OpCodeAsmJs::LdSlotArr:

Output::Print(L" R%d = R%d[%d] ", data->Value, data->Instance, data->SlotIndex);

break;

case OpCodeAsmJs::LdArr\_Func:

Output::Print(L" R%d = R%d[I%d] ", data->Value, data->Instance, data->SlotIndex);

break;

case OpCodeAsmJs::StSlot\_Int:

Output::Print(L" R%d[%d] = I%d ", data->Instance, data->SlotIndex, data->Value);

break;

case OpCodeAsmJs::StSlot\_Flt:

Output::Print(L" R%d[%d] = F%d ", data->Instance, data->SlotIndex, data->Value);

break;

case OpCodeAsmJs::StSlot\_Db:

Output::Print(L" R%d[%d] = D%d ", data->Instance, data->SlotIndex, data->Value);

break;

case OpCodeAsmJs::LdSlot\_Int:

Output::Print(L" I%d = R%d[%d] ", data->Value, data->Instance, data->SlotIndex);

break;

case OpCodeAsmJs::LdSlot\_Flt:

Output::Print(L" F%d = R%d[%d] ", data->Value, data->Instance, data->SlotIndex);

break;

case OpCodeAsmJs::LdSlot\_Db:

Output::Print(L" D%d = R%d[%d] ", data->Value, data->Instance, data->SlotIndex);

break;

case OpCodeAsmJs::Simd128\_LdSlot\_F4:

Output::Print(L" F4\_%d = R%d[%d] ", data->Value, data->Instance, data->SlotIndex);

break;

case OpCodeAsmJs::Simd128\_LdSlot\_I4:

Output::Print(L" I4\_%d = R%d[%d] ", data->Value, data->Instance, data->SlotIndex);

break;

case OpCodeAsmJs::Simd128\_LdSlot\_D2:

Output::Print(L" D2\_%d = R%d[%d] ", data->Value, data->Instance, data->SlotIndex);

break;

case OpCodeAsmJs::Simd128\_StSlot\_F4:

Output::Print(L" R%d[%d] = F4\_%d", data->Instance, data->SlotIndex, data->Value);

break;

case OpCodeAsmJs::Simd128\_StSlot\_I4:

Output::Print(L" R%d[%d] = I4\_%d", data->Instance, data->SlotIndex, data->Value);

break;

case OpCodeAsmJs::Simd128\_StSlot\_D2:

Output::Print(L" R%d[%d] = D2\_%d", data->Instance, data->SlotIndex, data->Value);

break;

default:

{

AssertMsg(false, "Unknown OpCode for OpLayoutElementSlot");

break;

}

}

}

template <class T>

void AsmJsByteCodeDumper::DumpAsmTypedArr(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

wchar\_t\* heapTag = nullptr;

wchar\_t valueTag = 'I';

switch (data->ViewType)

{

case ArrayBufferView::TYPE\_INT8:

heapTag = L"HEAP8"; valueTag = 'I'; break;

case ArrayBufferView::TYPE\_UINT8:

heapTag = L"HEAPU8"; valueTag = 'U'; break;

case ArrayBufferView::TYPE\_INT16:

heapTag = L"HEAP16"; valueTag = 'I'; break;

case ArrayBufferView::TYPE\_UINT16:

heapTag = L"HEAPU16"; valueTag = 'U'; break;

case ArrayBufferView::TYPE\_INT32:

heapTag = L"HEAP32"; valueTag = 'I'; break;

case ArrayBufferView::TYPE\_UINT32:

heapTag = L"HEAPU32"; valueTag = 'U'; break;

case ArrayBufferView::TYPE\_FLOAT32:

heapTag = L"HEAPF32"; valueTag = 'F'; break;

case ArrayBufferView::TYPE\_FLOAT64:

heapTag = L"HEAPF64"; valueTag = 'D'; break;

default:

Assert(false);

\_\_assume(false);

break;

}

switch (op)

{

case OpCodeAsmJs::LdArr:

Output::Print(L"%c%d = %s[I%d]", valueTag, data->Value, heapTag, data->SlotIndex); break;

case OpCodeAsmJs::LdArrConst:

Output::Print(L"%c%d = %s[%d]", valueTag, data->Value, heapTag, data->SlotIndex); break;

case OpCodeAsmJs::StArr:

Output::Print(L"%s[I%d] = %c%d", heapTag, data->SlotIndex, valueTag, data->Value); break;

case OpCodeAsmJs::StArrConst:

Output::Print(L"%s[%d] = %c%d", heapTag, data->SlotIndex, valueTag, data->Value); break;

default:

Assert(false);

\_\_assume(false);

break;

}

}

void AsmJsByteCodeDumper::DumpStartCall(OpCodeAsmJs op, const unaligned OpLayoutStartCall\* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

Assert(op == OpCodeAsmJs::StartCall || op == OpCodeAsmJs::I\_StartCall);

Output::Print(L" ArgSize: %d", data->ArgCount);

}

template <class T>

void AsmJsByteCodeDumper::DumpAsmCall(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

if (data->Return != Constants::NoRegister)

{

DumpReg((RegSlot)data->Return);

Output::Print(L"=");

}

Output::Print(L" R%d(ArgCount: %d)", data->Function, data->ArgCount);

}

template <class T>

void AsmJsByteCodeDumper::DumpAsmUnsigned1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpU4(data->C1);

}

void AsmJsByteCodeDumper::DumpEmpty(OpCodeAsmJs op, const unaligned OpLayoutEmpty\* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

// empty

}

void AsmJsByteCodeDumper::DumpAsmBr(OpCodeAsmJs op, const unaligned OpLayoutAsmBr\* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpOffset(data->RelativeJumpOffset, reader);

}

template <class T>

void AsmJsByteCodeDumper::DumpAsmReg1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg(data->R0);

}

template <class T>

void AsmJsByteCodeDumper::DumpAsmReg2(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg(data->R0);

DumpReg(data->R1);

}

template <class T>

void AsmJsByteCodeDumper::DumpAsmReg3(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg(data->R0);

DumpReg(data->R1);

DumpReg(data->R2);

}

template <class T>

void AsmJsByteCodeDumper::DumpAsmReg4(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg(data->R0);

DumpReg(data->R1);

DumpReg(data->R2);

DumpReg(data->R3);

}

template <class T>

void AsmJsByteCodeDumper::DumpAsmReg5(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg(data->R0);

DumpReg(data->R1);

DumpReg(data->R2);

DumpReg(data->R3);

DumpReg(data->R4);

}

template <class T>

void AsmJsByteCodeDumper::DumpAsmReg6(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg(data->R0);

DumpReg(data->R1);

DumpReg(data->R2);

DumpReg(data->R3);

DumpReg(data->R4);

DumpReg(data->R5);

}

template <class T>

void AsmJsByteCodeDumper::DumpAsmReg7(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg(data->R0);

DumpReg(data->R1);

DumpReg(data->R2);

DumpReg(data->R3);

DumpReg(data->R4);

DumpReg(data->R5);

DumpReg(data->R6);

}

template <class T>

void AsmJsByteCodeDumper::DumpAsmReg2IntConst1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg(data->R0);

DumpReg(data->R1);

DumpI4(data->C2);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt1Double1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpIntReg(data->I0);

DumpDoubleReg(data->D1);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt1Float1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpIntReg(data->I0);

DumpFloatReg(data->F1);

}

template <class T>

void AsmJsByteCodeDumper::DumpDouble1Int1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpDoubleReg(data->D0);

DumpIntReg(data->I1);

}

template <class T>

void AsmJsByteCodeDumper::DumpDouble1Float1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpDoubleReg(data->D0);

DumpFloatReg(data->F1);

}

template <class T>

void AsmJsByteCodeDumper::DumpDouble1Reg1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpDoubleReg(data->D0);

DumpReg(data->R1);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat1Reg1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloatReg(data->F0);

DumpReg(data->R1);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt1Reg1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpIntReg(data->I0);

DumpReg(data->R1);

}

template <class T>

void AsmJsByteCodeDumper::DumpReg1Double1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg(data->R0);

DumpDoubleReg(data->D1);

}

template <class T>

void AsmJsByteCodeDumper::DumpReg1Float1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg(data->R0);

DumpFloatReg(data->F1);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt1Double2(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpIntReg(data->I0);

DumpDoubleReg(data->D1);

DumpDoubleReg(data->D2);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt1Float2(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpIntReg(data->I0);

DumpFloatReg(data->F1);

DumpFloatReg(data->F2);

}

template <class T>

void AsmJsByteCodeDumper::DumpReg1Int1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg(data->R0);

DumpIntReg(data->I1);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt2(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpIntReg(data->I0);

DumpIntReg(data->I1);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt1Const1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpIntReg(data->I0);

DumpI4(data->C1);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt3(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpIntReg(data->I0);

DumpIntReg(data->I1);

DumpIntReg(data->I2);

}

template <class T>

void AsmJsByteCodeDumper::DumpDouble2(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpDoubleReg(data->D0);

DumpDoubleReg(data->D1);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat2(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloatReg(data->F0);

DumpFloatReg(data->F1);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat3(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloatReg(data->F0);

DumpFloatReg(data->F1);

DumpFloatReg(data->F2);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat1Double1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloatReg(data->F0);

DumpDoubleReg(data->D1);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat1Int1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloatReg(data->F0);

DumpIntReg(data->I1);

}

template <class T>

void AsmJsByteCodeDumper::DumpDouble3(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpDoubleReg(data->D0);

DumpDoubleReg(data->D1);

DumpDoubleReg(data->D2);

}

template <class T>

void AsmJsByteCodeDumper::DumpBrInt1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpOffset(data->RelativeJumpOffset, reader);

DumpIntReg(data->I1);

}

template <class T>

void AsmJsByteCodeDumper::DumpBrInt2(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpOffset(data->RelativeJumpOffset, reader);

DumpIntReg(data->I1);

DumpIntReg(data->I2);

}

// Float32x4

template <class T>

void AsmJsByteCodeDumper::DumpFloat32x4\_2(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat32x4Reg(data->F4\_0);

DumpFloat32x4Reg(data->F4\_1);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat32x4\_3(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat32x4Reg(data->F4\_0);

DumpFloat32x4Reg(data->F4\_1);

DumpFloat32x4Reg(data->F4\_2);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat32x4\_4(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat32x4Reg(data->F4\_0);

DumpFloat32x4Reg(data->F4\_1);

DumpFloat32x4Reg(data->F4\_2);

DumpFloat32x4Reg(data->F4\_3);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat32x4\_1Float4(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat32x4Reg(data->F4\_0);

DumpFloatReg(data->F1);

DumpFloatReg(data->F2);

DumpFloatReg(data->F3);

DumpFloatReg(data->F4);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat32x4\_2Int4(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat32x4Reg(data->F4\_0);

DumpFloat32x4Reg(data->F4\_1);

DumpIntReg(data->I2);

DumpIntReg(data->I3);

DumpIntReg(data->I4);

DumpIntReg(data->I5);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat32x4\_3Int4(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat32x4Reg(data->F4\_0);

DumpFloat32x4Reg(data->F4\_1);

DumpFloat32x4Reg(data->F4\_2);

DumpIntReg(data->I3);

DumpIntReg(data->I4);

DumpIntReg(data->I5);

DumpIntReg(data->I6);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat32x4\_1Float1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat32x4Reg(data->F4\_0);

DumpFloatReg(data->F1);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat32x4\_2Float1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat32x4Reg(data->F4\_0);

DumpFloat32x4Reg(data->F4\_1);

DumpFloatReg(data->F2);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat32x4\_1Float64x2\_1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat32x4Reg(data->F4\_0);

DumpFloat64x2Reg(data->D2\_1);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat32x4\_1Int32x4\_1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat32x4Reg(data->F4\_0);

DumpInt32x4Reg(data->I4\_1);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat32x4\_1Int32x4\_1Float32x4\_2(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat32x4Reg(data->F4\_0);

DumpInt32x4Reg(data->I4\_1);

DumpFloat32x4Reg(data->F4\_2);

DumpFloat32x4Reg(data->F4\_3);

}

template <class T>

void AsmJsByteCodeDumper::DumpReg1Float32x4\_1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg(data->R0);

DumpFloat32x4Reg(data->F4\_1);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt1Float32x4\_1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpIntReg(data->I0);

DumpFloat32x4Reg(data->F4\_1);

}

// Int32x4

template <class T>

void AsmJsByteCodeDumper::DumpInt32x4\_2(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpInt32x4Reg(data->I4\_0);

DumpInt32x4Reg(data->I4\_1);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt32x4\_3(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpInt32x4Reg(data->I4\_0);

DumpInt32x4Reg(data->I4\_1);

DumpInt32x4Reg(data->I4\_2);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt32x4\_4(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpInt32x4Reg(data->I4\_0);

DumpInt32x4Reg(data->I4\_1);

DumpInt32x4Reg(data->I4\_2);

DumpInt32x4Reg(data->I4\_3);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt32x4\_1Int4(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpInt32x4Reg(data->I4\_0);

DumpIntReg(data->I1);

DumpIntReg(data->I2);

DumpIntReg(data->I3);

DumpIntReg(data->I4);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt32x4\_2Int4(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpInt32x4Reg(data->I4\_0);

DumpInt32x4Reg(data->I4\_1);

DumpIntReg(data->I2);

DumpIntReg(data->I3);

DumpIntReg(data->I4);

DumpIntReg(data->I5);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt32x4\_2Int2(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpInt32x4Reg(data->I4\_0);

DumpInt32x4Reg(data->I4\_1);

DumpIntReg(data->I2);

DumpIntReg(data->I3);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt1Int32x4\_1Int1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpIntReg(data->I0);

DumpInt32x4Reg(data->I4\_1);

DumpIntReg(data->I2);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat32x4\_2Int1Float1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat32x4Reg(data->F4\_0);

DumpFloat32x4Reg(data->F4\_1);

DumpIntReg(data->I2);

DumpFloatReg(data->F3);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat1Float32x4\_1Int1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloatReg(data->F0);

DumpInt32x4Reg(data->F4\_1);

DumpIntReg(data->I2);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt32x4\_3Int4(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpInt32x4Reg(data->I4\_0);

DumpInt32x4Reg(data->I4\_1);

DumpInt32x4Reg(data->I4\_2);

DumpIntReg(data->I3);

DumpIntReg(data->I4);

DumpIntReg(data->I5);

DumpIntReg(data->I6);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt32x4\_1Int1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpInt32x4Reg(data->I4\_0);

DumpIntReg(data->I1);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt32x4\_2Int1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpInt32x4Reg(data->I4\_0);

DumpInt32x4Reg(data->I4\_1);

DumpIntReg(data->I2);

}

template <class T>

void AsmJsByteCodeDumper::DumpReg1Int32x4\_1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg(data->R0);

DumpInt32x4Reg(data->I4\_1);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt32x4\_1Float32x4\_1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpInt32x4Reg(data->I4\_0);

DumpFloat32x4Reg(data->F4\_1);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt32x4\_1Float64x2\_1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpInt32x4Reg(data->I4\_0);

DumpFloat64x2Reg(data->D2\_1);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt1Int32x4\_1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpIntReg(data->I0);

DumpFloat64x2Reg(data->I4\_1);

}

// Float64x2

template <class T>

void AsmJsByteCodeDumper::DumpFloat64x2\_2(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat64x2Reg(data->D2\_0);

DumpFloat64x2Reg(data->D2\_1);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat64x2\_3(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat64x2Reg(data->D2\_0);

DumpFloat64x2Reg(data->D2\_1);

DumpFloat64x2Reg(data->D2\_2);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat64x2\_4(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat64x2Reg(data->D2\_0);

DumpFloat64x2Reg(data->D2\_1);

DumpFloat64x2Reg(data->D2\_2);

DumpFloat64x2Reg(data->D2\_3);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat64x2\_1Double2(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat64x2Reg(data->D2\_0);

DumpDoubleReg(data->D1);

DumpDoubleReg(data->D2);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat64x2\_1Double1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat64x2Reg(data->D2\_0);

DumpDoubleReg(data->D1);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat64x2\_2Double1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat64x2Reg(data->D2\_0);

DumpFloat64x2Reg(data->D2\_1);

DumpDoubleReg(data->D2);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat64x2\_2Int2(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat64x2Reg(data->D2\_0);

DumpFloat64x2Reg(data->D2\_1);

DumpIntReg(data->I2);

DumpIntReg(data->I3);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat64x2\_3Int2(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat64x2Reg(data->D2\_0);

DumpFloat64x2Reg(data->D2\_1);

DumpFloat64x2Reg(data->D2\_2);

DumpIntReg(data->I3);

DumpIntReg(data->I4);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat64x2\_1Float32x4\_1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat64x2Reg(data->D2\_0);

DumpFloat32x4Reg(data->F4\_1);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat64x2\_1Int32x4\_1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat64x2Reg(data->D2\_0);

DumpInt32x4Reg(data->I4\_1);

}

template <class T>

void AsmJsByteCodeDumper::DumpFloat64x2\_1Int32x4\_1Float64x2\_2(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpFloat64x2Reg(data->D2\_0);

DumpInt32x4Reg(data->I4\_1);

DumpFloat64x2Reg(data->D2\_2);

DumpFloat64x2Reg(data->D2\_3);

}

template <class T>

void AsmJsByteCodeDumper::DumpReg1Float64x2\_1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg(data->R0);

DumpFloat64x2Reg(data->D2\_1);

}

template <class T>

void AsmJsByteCodeDumper::DumpInt1Float64x2\_1(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpIntReg(data->I0);

DumpFloat64x2Reg(data->D2\_1);

}

template <class T>

void AsmJsByteCodeDumper::DumpAsmSimdTypedArr(OpCodeAsmJs op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

wchar\_t\* heapTag = nullptr;

switch (data->ViewType)

{

case ArrayBufferView::TYPE\_INT8:

heapTag = L"HEAP8"; break;

case ArrayBufferView::TYPE\_UINT8:

heapTag = L"HEAPU8"; break;

case ArrayBufferView::TYPE\_INT16:

heapTag = L"HEAP16"; break;

case ArrayBufferView::TYPE\_UINT16:

heapTag = L"HEAPU16"; break;

case ArrayBufferView::TYPE\_INT32:

heapTag = L"HEAP32"; break;

case ArrayBufferView::TYPE\_UINT32:

heapTag = L"HEAPU32"; break;

case ArrayBufferView::TYPE\_FLOAT32:

heapTag = L"HEAPF32"; break;

case ArrayBufferView::TYPE\_FLOAT64:

heapTag = L"HEAPF64"; break;

default:

Assert(false);

\_\_assume(false);

break;

}

switch (op)

{

case OpCodeAsmJs::Simd128\_LdArrConst\_I4:

case OpCodeAsmJs::Simd128\_LdArrConst\_F4:

case OpCodeAsmJs::Simd128\_LdArrConst\_D2:

case OpCodeAsmJs::Simd128\_StArrConst\_I4:

case OpCodeAsmJs::Simd128\_StArrConst\_F4:

case OpCodeAsmJs::Simd128\_StArrConst\_D2:

Output::Print(L" %s[%d] ", heapTag, data->SlotIndex);

break;

case OpCodeAsmJs::Simd128\_LdArr\_I4:

case OpCodeAsmJs::Simd128\_LdArr\_F4:

case OpCodeAsmJs::Simd128\_LdArr\_D2:

case OpCodeAsmJs::Simd128\_StArr\_I4:

case OpCodeAsmJs::Simd128\_StArr\_F4:

case OpCodeAsmJs::Simd128\_StArr\_D2:

Output::Print(L" %s[I%d] ", heapTag, data->SlotIndex);

break;

default:

Assert(false);

\_\_assume(false);

break;

}

switch (op)

{

case OpCodeAsmJs::Simd128\_LdArr\_I4:

case OpCodeAsmJs::Simd128\_LdArrConst\_I4:

case OpCodeAsmJs::Simd128\_StArr\_I4:

case OpCodeAsmJs::Simd128\_StArrConst\_I4:

DumpInt32x4Reg(data->Value);

break;

case OpCodeAsmJs::Simd128\_LdArr\_F4:

case OpCodeAsmJs::Simd128\_LdArrConst\_F4:

case OpCodeAsmJs::Simd128\_StArr\_F4:

case OpCodeAsmJs::Simd128\_StArrConst\_F4:

DumpFloat32x4Reg(data->Value);

break;

case OpCodeAsmJs::Simd128\_LdArr\_D2:

case OpCodeAsmJs::Simd128\_LdArrConst\_D2:

case OpCodeAsmJs::Simd128\_StArr\_D2:

case OpCodeAsmJs::Simd128\_StArrConst\_D2:

DumpFloat64x2Reg(data->Value);

break;

default:

Assert(false);

\_\_assume(false);

break;

}

// data width

Output::Print(L" %d bytes ", data->DataWidth);

}

}

#endif

#endif

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

// 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

#ifndef TEMP\_DISABLE\_ASMJS

namespace Js {

#if DBG\_DUMP

class AsmJsByteCodeDumper : public ByteCodeDumper

{

public:

static void Dump(AsmJsFunc\* func, FunctionBody\* body);

static void DumpConstants(AsmJsFunc\* func, FunctionBody\* body);

static void DumpOp(OpCodeAsmJs op, LayoutSize layoutSize, ByteCodeReader& reader, FunctionBody\* dumpFunction);

static void DumpIntReg(RegSlot reg);

static void DumpDoubleReg(RegSlot reg);

static void DumpFloatReg(RegSlot reg);

static void DumpR8Float(float value);

static void DumpFloat32x4Reg(RegSlot reg);

static void DumpInt32x4Reg(RegSlot reg);

static void DumpFloat64x2Reg(RegSlot reg);

#define LAYOUT\_TYPE(layout) \

static void Dump##layout(OpCodeAsmJs op, const unaligned OpLayout##layout\* data, FunctionBody \* dumpFunction, ByteCodeReader& reader);

#define LAYOUT\_TYPE\_WMS(layout) \

template <class T> static void Dump##layout(OpCodeAsmJs op, const unaligned T\* data, FunctionBody \* dumpFunction, ByteCodeReader& reader);

#include "LayoutTypesAsmJs.h"

};

#endif

}

#endif

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

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

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

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

#include "RuntimeByteCodePch.h"

#ifndef TEMP\_DISABLE\_ASMJS

#include "ByteCode\AsmJsByteCodeWriter.h"

namespace Js

{

template <>

\_\_inline uint ByteCodeWriter::Data::EncodeT<SmallLayout>(OpCodeAsmJs op, ByteCodeWriter\* writer, bool isPatching)

{

Assert(op < Js::OpCodeAsmJs::ByteCodeLast);

uint offset;

if (op <= Js::OpCode::MaxByteSizedOpcodes)

{

byte byteop = (byte)op;

offset = Write(&byteop, sizeof(byte));

}

else

{

byte byteop = (byte)Js::OpCodeAsmJs::ExtendedOpcodePrefix;

offset = Write(&byteop, sizeof(byte));

byteop = (byte)op;

Write(&byteop, sizeof(byte));

}

if (!isPatching)

{

writer->IncreaseByteCodeCount();

}

return offset;

}

template <LayoutSize layoutSize>

\_\_inline uint ByteCodeWriter::Data::EncodeT(OpCodeAsmJs op, ByteCodeWriter\* writer, bool isPatching)

{

Assert(op < Js::OpCodeAsmJs::ByteCodeLast);

CompileAssert(layoutSize != SmallLayout);

const byte exop = (byte)((op <= Js::OpCodeAsmJs::MaxByteSizedOpcodes) ?

(layoutSize == LargeLayout ? Js::OpCodeAsmJs::LargeLayoutPrefix : Js::OpCodeAsmJs::MediumLayoutPrefix) :

(layoutSize == LargeLayout ? Js::OpCodeAsmJs::ExtendedLargeLayoutPrefix : Js::OpCodeAsmJs::ExtendedMediumLayoutPrefix));

uint offset = Write(&exop, sizeof(byte));

byte byteop = (byte)op;

Write(&byteop, sizeof(byte));

if (!isPatching)

{

writer->IncreaseByteCodeCount();

}

return offset;

}

template <LayoutSize layoutSize>

\_\_inline uint ByteCodeWriter::Data::EncodeT(OpCodeAsmJs op, const void\* rawData, int byteSize, ByteCodeWriter\* writer, bool isPatching)

{

AssertMsg((rawData != nullptr) && (byteSize < 100), "Ensure valid data for opcode");

uint offset = EncodeT<layoutSize>(op, writer, isPatching);

Write(rawData, byteSize);

return offset;

}

void AsmJsByteCodeWriter::InitData(ArenaAllocator\* alloc, long initCodeBufferSize)

{

ByteCodeWriter::InitData(alloc, initCodeBufferSize);

#ifdef BYTECODE\_BRANCH\_ISLAND

useBranchIsland = false;

#endif

}

#define MULTISIZE\_LAYOUT\_WRITE(layout, ...) \

if (!TryWrite##layout<SmallLayoutSizePolicy>(\_\_VA\_ARGS\_\_) && !TryWrite##layout<MediumLayoutSizePolicy>(\_\_VA\_ARGS\_\_)) \

{ \

bool success = TryWrite##layout<LargeLayoutSizePolicy>(\_\_VA\_ARGS\_\_); \

Assert(success); \

}

//////////////////////////////////////////////////////////////////////////

/// Asm.js Specific functions

template <typename SizePolicy>

bool AsmJsByteCodeWriter::TryWriteAsmJsUnsigned1(OpCodeAsmJs op, uint C1)

{

OpLayoutT\_AsmUnsigned1<SizePolicy> layout;

if (SizePolicy::Assign(layout.C1, C1))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

template <typename SizePolicy>

bool AsmJsByteCodeWriter::TryWriteAsmReg1(OpCodeAsmJs op, RegSlot R0)

{

OpLayoutT\_AsmReg1<SizePolicy> layout;

if (SizePolicy::Assign(layout.R0, R0))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

template <typename SizePolicy>

bool AsmJsByteCodeWriter::TryWriteAsmReg2(OpCodeAsmJs op, RegSlot R0, RegSlot R1)

{

OpLayoutT\_AsmReg2<SizePolicy> layout;

if (SizePolicy::Assign(layout.R0, R0) && SizePolicy::Assign(layout.R1, R1))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

template <typename SizePolicy>

bool AsmJsByteCodeWriter::TryWriteAsmReg3(OpCodeAsmJs op, RegSlot R0, RegSlot R1, RegSlot R2)

{

OpLayoutT\_AsmReg3<SizePolicy> layout;

if (SizePolicy::Assign(layout.R0, R0) && SizePolicy::Assign(layout.R1, R1) && SizePolicy::Assign(layout.R2, R2))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

template <typename SizePolicy>

bool AsmJsByteCodeWriter::TryWriteAsmReg4(OpCodeAsmJs op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3)

{

OpLayoutT\_AsmReg4<SizePolicy> layout;

if (SizePolicy::Assign(layout.R0, R0) && SizePolicy::Assign(layout.R1, R1) && SizePolicy::Assign(layout.R2, R2) && SizePolicy::Assign(layout.R3, R3))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

template <typename SizePolicy>

bool AsmJsByteCodeWriter::TryWriteAsmReg5(OpCodeAsmJs op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3, RegSlot R4)

{

OpLayoutT\_AsmReg5<SizePolicy> layout;

if (SizePolicy::Assign(layout.R0, R0) && SizePolicy::Assign(layout.R1, R1) && SizePolicy::Assign(layout.R2, R2) && SizePolicy::Assign(layout.R3, R3) && SizePolicy::Assign(layout.R4, R4))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

template <typename SizePolicy>

bool AsmJsByteCodeWriter::TryWriteAsmReg6(OpCodeAsmJs op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3, RegSlot R4, RegSlot R5)

{

OpLayoutT\_AsmReg6<SizePolicy> layout;

if (SizePolicy::Assign(layout.R0, R0) && SizePolicy::Assign(layout.R1, R1) && SizePolicy::Assign(layout.R2, R2) && SizePolicy::Assign(layout.R3, R3) && SizePolicy::Assign(layout.R4, R4) && SizePolicy::Assign(layout.R5, R5))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

template <typename SizePolicy>

bool AsmJsByteCodeWriter::TryWriteAsmReg7(OpCodeAsmJs op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3, RegSlot R4, RegSlot R5, RegSlot R6)

{

OpLayoutT\_AsmReg7<SizePolicy> layout;

if (SizePolicy::Assign(layout.R0, R0) && SizePolicy::Assign(layout.R1, R1) && SizePolicy::Assign(layout.R2, R2) && SizePolicy::Assign(layout.R3, R3) && SizePolicy::Assign(layout.R4, R4) && SizePolicy::Assign(layout.R5, R5) && SizePolicy::Assign(layout.R6, R6))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

template <typename SizePolicy>

bool AsmJsByteCodeWriter::TryWriteAsmReg2IntConst1(OpCodeAsmJs op, RegSlot R0, RegSlot R1, int C2)

{

OpLayoutT\_AsmReg2IntConst1<SizePolicy> layout;

if (SizePolicy::Assign(layout.R0, R0) && SizePolicy::Assign(layout.R1, R1) && SizePolicy::Assign(layout.C2, C2))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

template <typename SizePolicy>

bool AsmJsByteCodeWriter::TryWriteInt1Const1(OpCodeAsmJs op, RegSlot R0, int C1)

{

OpLayoutT\_Int1Const1<SizePolicy> layout;

if (SizePolicy::Assign(layout.I0, R0) && SizePolicy::Assign(layout.C1, C1))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

template <typename SizePolicy>

bool AsmJsByteCodeWriter::TryWriteAsmBrReg1(OpCodeAsmJs op, ByteCodeLabel labelID, RegSlot R1)

{

OpLayoutT\_BrInt1<SizePolicy> layout;

if (SizePolicy::Assign(layout.I1, R1))

{

size\_t const offsetOfRelativeJumpOffsetFromEnd = sizeof(OpLayoutT\_BrInt1<SizePolicy>) - offsetof(OpLayoutT\_BrInt1<SizePolicy>, RelativeJumpOffset);

layout.RelativeJumpOffset = offsetOfRelativeJumpOffsetFromEnd;

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

AddJumpOffset(op, labelID, offsetOfRelativeJumpOffsetFromEnd);

return true;

}

return false;

}

template <typename SizePolicy>

bool AsmJsByteCodeWriter::TryWriteAsmBrReg2(OpCodeAsmJs op, ByteCodeLabel labelID, RegSlot R1, RegSlot R2)

{

OpLayoutT\_BrInt2<SizePolicy> layout;

if (SizePolicy::Assign(layout.I1, R1) && SizePolicy::Assign(layout.I2, R2))

{

size\_t const offsetOfRelativeJumpOffsetFromEnd = sizeof(OpLayoutT\_BrInt2<SizePolicy>) - offsetof(OpLayoutT\_BrInt2<SizePolicy>, RelativeJumpOffset);

layout.RelativeJumpOffset = offsetOfRelativeJumpOffsetFromEnd;

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

AddJumpOffset(op, labelID, offsetOfRelativeJumpOffsetFromEnd);

return true;

}

return false;

}

template <typename SizePolicy>

bool AsmJsByteCodeWriter::TryWriteAsmCall(OpCodeAsmJs op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, AsmJsRetType retType)

{

OpLayoutT\_AsmCall<SizePolicy> layout;

if (SizePolicy::Assign(layout.Return, returnValueRegister) && SizePolicy::Assign(layout.Function, functionRegister)

&& SizePolicy::Assign(layout.ArgCount, givenArgCount) && SizePolicy::Assign<int8>(layout.ReturnType, (int8)retType.which()))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

template <typename SizePolicy>

bool AsmJsByteCodeWriter::TryWriteAsmSlot(OpCodeAsmJs op, RegSlot value, RegSlot instance, int32 slotId)

{

OpLayoutT\_ElementSlot<SizePolicy> layout;

if (SizePolicy::Assign(layout.Value, value) && SizePolicy::Assign(layout.Instance, instance)

&& SizePolicy::Assign(layout.SlotIndex, slotId))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

template <typename SizePolicy>

bool AsmJsByteCodeWriter::TryWriteAsmTypedArr(OpCodeAsmJs op, RegSlot value, uint32 slotIndex, ArrayBufferView::ViewType viewType)

{

OpLayoutT\_AsmTypedArr<SizePolicy> layout;

if (SizePolicy::Assign(layout.Value, value) && SizePolicy::Assign<int8>(layout.ViewType, (int8)viewType)

&& SizePolicy::Assign(layout.SlotIndex, slotIndex))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

template <typename SizePolicy>

bool AsmJsByteCodeWriter::TryWriteAsmSimdTypedArr(OpCodeAsmJs op, RegSlot value, uint32 slotIndex, uint8 dataWidth, ArrayBufferView::ViewType viewType)

{

OpLayoutT\_AsmSimdTypedArr<SizePolicy> layout;

if (SizePolicy::Assign(layout.Value, value) && SizePolicy::Assign<int8>(layout.ViewType, (int8)viewType)

&& SizePolicy::Assign(layout.SlotIndex, slotIndex) && SizePolicy::Assign<int8>(layout.DataWidth, dataWidth))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void AsmJsByteCodeWriter::EmptyAsm(OpCodeAsmJs op)

{

m\_byteCodeData.Encode(op, this);

}

void AsmJsByteCodeWriter::Conv(OpCodeAsmJs op, RegSlot R0, RegSlot R1)

{

MULTISIZE\_LAYOUT\_WRITE(AsmReg2, op, R0, R1);

}

void AsmJsByteCodeWriter::AsmInt1Const1(OpCodeAsmJs op, RegSlot R0, int C1)

{

MULTISIZE\_LAYOUT\_WRITE(Int1Const1, op, R0, C1);

}

void AsmJsByteCodeWriter::AsmReg1(OpCodeAsmJs op, RegSlot R0)

{

MULTISIZE\_LAYOUT\_WRITE(AsmReg1, op, R0);

}

void AsmJsByteCodeWriter::AsmReg2(OpCodeAsmJs op, RegSlot R0, RegSlot R1)

{

MULTISIZE\_LAYOUT\_WRITE(AsmReg2, op, R0, R1);

}

void AsmJsByteCodeWriter::AsmReg3(OpCodeAsmJs op, RegSlot R0, RegSlot R1, RegSlot R2)

{

MULTISIZE\_LAYOUT\_WRITE(AsmReg3, op, R0, R1, R2);

}

void AsmJsByteCodeWriter::AsmReg4(OpCodeAsmJs op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3)

{

MULTISIZE\_LAYOUT\_WRITE(AsmReg4, op, R0, R1, R2, R3);

}

void AsmJsByteCodeWriter::AsmReg5(OpCodeAsmJs op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3, RegSlot R4)

{

MULTISIZE\_LAYOUT\_WRITE(AsmReg5, op, R0, R1, R2, R3, R4);

}

void AsmJsByteCodeWriter::AsmReg6(OpCodeAsmJs op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3, RegSlot R4, RegSlot R5)

{

MULTISIZE\_LAYOUT\_WRITE(AsmReg6, op, R0, R1, R2, R3, R4, R5);

}

void AsmJsByteCodeWriter::AsmReg7(OpCodeAsmJs op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3, RegSlot R4, RegSlot R5, RegSlot R6)

{

MULTISIZE\_LAYOUT\_WRITE(AsmReg7, op, R0, R1, R2, R3, R4, R5, R6);

}

void AsmJsByteCodeWriter::AsmReg2IntConst1(OpCodeAsmJs op, RegSlot R0, RegSlot R1, int C2)

{

MULTISIZE\_LAYOUT\_WRITE(AsmReg2IntConst1, op, R0, R1, C2);

}

void AsmJsByteCodeWriter::AsmBr(ByteCodeLabel labelID, OpCodeAsmJs op)

{

CheckOpen();

CheckLabel(labelID);

size\_t const offsetOfRelativeJumpOffsetFromEnd = sizeof(OpLayoutAsmBr) - offsetof(OpLayoutAsmBr, RelativeJumpOffset);

OpLayoutAsmBr data;

data.RelativeJumpOffset = offsetOfRelativeJumpOffsetFromEnd;

m\_byteCodeData.Encode(op, &data, sizeof(data), this);

AddJumpOffset(op, labelID, offsetOfRelativeJumpOffsetFromEnd);

}

void AsmJsByteCodeWriter::AsmBrReg1(OpCodeAsmJs op, ByteCodeLabel labelID, RegSlot R1)

{

CheckOpen();

CheckLabel(labelID);

MULTISIZE\_LAYOUT\_WRITE(AsmBrReg1, op, labelID, R1);

}

void AsmJsByteCodeWriter::AsmBrReg2(OpCodeAsmJs op, ByteCodeLabel labelID, RegSlot R1, RegSlot R2)

{

CheckOpen();

CheckLabel(labelID);

MULTISIZE\_LAYOUT\_WRITE(AsmBrReg2, op, labelID, R1, R2);

}

void AsmJsByteCodeWriter::AsmStartCall(OpCodeAsmJs op, ArgSlot ArgCount, bool isPatching)

{

CheckOpen();

OpLayoutStartCall data;

data.ArgCount = ArgCount;

m\_byteCodeData.Encode(op, &data, sizeof(data), this, isPatching);

}

void AsmJsByteCodeWriter::AsmCall(OpCodeAsmJs op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, AsmJsRetType retType)

{

MULTISIZE\_LAYOUT\_WRITE(AsmCall, op, returnValueRegister, functionRegister, givenArgCount, retType);

}

void AsmJsByteCodeWriter::AsmTypedArr(OpCodeAsmJs op, RegSlot value, uint32 slotIndex, ArrayBufferView::ViewType viewType)

{

MULTISIZE\_LAYOUT\_WRITE(AsmTypedArr, op, value, slotIndex, viewType);

}

void AsmJsByteCodeWriter::AsmSimdTypedArr(OpCodeAsmJs op, RegSlot value, uint32 slotIndex, uint8 dataWidth, ArrayBufferView::ViewType viewType)

{

Assert(dataWidth >= 4 && dataWidth <= 16);

MULTISIZE\_LAYOUT\_WRITE(AsmSimdTypedArr, op, value, slotIndex, dataWidth, viewType);

}

void AsmJsByteCodeWriter::AsmSlot(OpCodeAsmJs op, RegSlot value, RegSlot instance, int32 slotId)

{

MULTISIZE\_LAYOUT\_WRITE(AsmSlot, op, value, instance, slotId);

}

uint AsmJsByteCodeWriter::EnterLoop(Js::ByteCodeLabel loopEntrance)

{

uint loopId = m\_functionWrite->IncrLoopCount();

Assert((uint)m\_loopHeaders->Count() == loopId);

m\_loopHeaders->Add(LoopHeaderData(m\_byteCodeData.GetCurrentOffset(), 0, m\_loopNest > 0));

m\_loopNest++;

Js::OpCodeAsmJs loopBodyOpcode = Js::OpCodeAsmJs::AsmJsLoopBodyStart;

this->MarkAsmJsLabel(loopEntrance);

this->AsmJsUnsigned1(loopBodyOpcode, loopId);

return loopId;

}

void AsmJsByteCodeWriter::AsmJsUnsigned1(OpCodeAsmJs op, uint c1)

{

MULTISIZE\_LAYOUT\_WRITE(AsmJsUnsigned1, op, c1);

}

void AsmJsByteCodeWriter::ExitLoop(uint loopId)

{

Assert(m\_loopNest > 0);

m\_loopNest--;

m\_loopHeaders->Item(loopId).endOffset = m\_byteCodeData.GetCurrentOffset();

}

void AsmJsByteCodeWriter::AddJumpOffset(Js::OpCodeAsmJs op, ByteCodeLabel labelId, uint fieldByteOffset)

{

AssertMsg(fieldByteOffset < 100, "Ensure valid field offset");

CheckOpen();

CheckLabel(labelId);

uint jumpByteOffset = m\_byteCodeData.GetCurrentOffset() - fieldByteOffset;

//

// Branch targets are created in two passes:

// - In the instruction stream, write "labelID" into "OpLayoutBrC.Offset". Record this

// location in "m\_jumpOffsets" to be patched later.

// - When the bytecode is closed, update all "OpLayoutBrC.Offset"'s with their actual

// destinations.

//

JumpInfo jumpInfo = { labelId, jumpByteOffset };

m\_jumpOffsets->Add(jumpInfo);

}

void AsmJsByteCodeWriter::MarkAsmJsLabel(ByteCodeLabel labelID)

{

MarkLabel(labelID);

EmptyAsm(OpCodeAsmJs::Label);

}

} // namespace Js

#endif

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

// 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

#ifndef TEMP\_DISABLE\_ASMJS

namespace Js

{

struct AsmJsByteCodeWriter : public ByteCodeWriter

{

private:

using ByteCodeWriter::MarkLabel;

public:

void InitData ( ArenaAllocator\* alloc, long initCodeBufferSize );

void EmptyAsm ( OpCodeAsmJs op );

void Conv ( OpCodeAsmJs op, RegSlot R0, RegSlot R1 );

void AsmInt1Const1 ( OpCodeAsmJs op, RegSlot R0, int C1 );

void AsmReg1 ( OpCodeAsmJs op, RegSlot R0 );

void AsmReg2 ( OpCodeAsmJs op, RegSlot R0, RegSlot R1 );

void AsmReg3 ( OpCodeAsmJs op, RegSlot R0, RegSlot R1, RegSlot R2 );

void AsmReg4 ( OpCodeAsmJs op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3 );

void AsmReg5 ( OpCodeAsmJs op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3, RegSlot R4 );

void AsmReg6 ( OpCodeAsmJs op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3, RegSlot R4, RegSlot R5 );

void AsmReg7 ( OpCodeAsmJs op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3, RegSlot R4, RegSlot R5, RegSlot R6 );

void AsmReg2IntConst1( OpCodeAsmJs op, RegSlot R0, RegSlot R1, int C2);

void AsmBr ( ByteCodeLabel labelID, OpCodeAsmJs op = OpCodeAsmJs::AsmBr );

void AsmBrReg1 ( OpCodeAsmJs op, ByteCodeLabel labelID, RegSlot R1 );

void AsmBrReg2 ( OpCodeAsmJs op, ByteCodeLabel labelID, RegSlot R1, RegSlot R2 );

void AsmStartCall ( OpCodeAsmJs op, ArgSlot ArgCount, bool isPatching = false);

void AsmCall ( OpCodeAsmJs op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, AsmJsRetType retType );

void AsmSlot ( OpCodeAsmJs op, RegSlot value, RegSlot instance, int32 slotId );

void AsmTypedArr ( OpCodeAsmJs op, RegSlot value, uint32 slotIndex, ArrayBufferView::ViewType viewType );

void AsmSimdTypedArr ( OpCodeAsmJs op, RegSlot value, uint32 slotIndex, uint8 dataWidth, ArrayBufferView::ViewType viewType );

void MarkAsmJsLabel ( ByteCodeLabel labelID );

void AsmJsUnsigned1 ( OpCodeAsmJs op, uint C1 );

uint EnterLoop ( ByteCodeLabel loopEntrance );

void ExitLoop ( uint loopId );

private:

template <typename SizePolicy> bool TryWriteAsmReg1 ( OpCodeAsmJs op, RegSlot R0 );

template <typename SizePolicy> bool TryWriteAsmReg2 ( OpCodeAsmJs op, RegSlot R0, RegSlot R1 );

template <typename SizePolicy> bool TryWriteAsmReg3 ( OpCodeAsmJs op, RegSlot R0, RegSlot R1, RegSlot R2 );

template <typename SizePolicy> bool TryWriteAsmReg4 ( OpCodeAsmJs op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3 );

template <typename SizePolicy> bool TryWriteAsmReg5 ( OpCodeAsmJs op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3, RegSlot R4 );

template <typename SizePolicy> bool TryWriteAsmReg6 ( OpCodeAsmJs op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3, RegSlot R4, RegSlot R5 );

template <typename SizePolicy> bool TryWriteAsmReg7 ( OpCodeAsmJs op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3, RegSlot R4, RegSlot R5, RegSlot R6 );

template <typename SizePolicy> bool TryWriteAsmReg2IntConst1( OpCodeAsmJs op, RegSlot R0, RegSlot R1, int C2 );

template <typename SizePolicy> bool TryWriteInt1Const1 ( OpCodeAsmJs op, RegSlot R0, int C1 );

template <typename SizePolicy> bool TryWriteAsmBrReg1 ( OpCodeAsmJs op, ByteCodeLabel labelID, RegSlot R1 );

template <typename SizePolicy> bool TryWriteAsmBrReg2 ( OpCodeAsmJs op, ByteCodeLabel labelID, RegSlot R1, RegSlot R2 );

template <typename SizePolicy> bool TryWriteAsmCall ( OpCodeAsmJs op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, AsmJsRetType retType );

template <typename SizePolicy> bool TryWriteAsmSlot ( OpCodeAsmJs op, RegSlot value, RegSlot instance, int32 slotId );

template <typename SizePolicy> bool TryWriteAsmTypedArr ( OpCodeAsmJs op, RegSlot value, uint32 slotIndex, ArrayBufferView::ViewType viewType );

template <typename SizePolicy> bool TryWriteAsmSimdTypedArr ( OpCodeAsmJs op, RegSlot value, uint32 slotIndex, uint8 dataWidth, ArrayBufferView::ViewType viewType );

template <typename SizePolicy> bool TryWriteAsmJsUnsigned1 ( OpCodeAsmJs op, uint C1 );

void AddJumpOffset( Js::OpCodeAsmJs op, ByteCodeLabel labelId, uint fieldByteOffset );

};

}

#endif

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

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

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

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

#pragma once

namespace Js

{

template<typename T>

struct AuxArray

{

uint32 count;

T elements[];

AuxArray(uint32 count) : count(count)

{

}

void SetCount(uint count) { this->count = count; }

size\_t GetDataSize() const { return sizeof(AuxArray) + sizeof(T) \* count; }

};

typedef AuxArray<Var> VarArray;

struct FuncInfoEntry

{

uint nestedIndex;

uint scopeSlot;

};

typedef AuxArray<FuncInfoEntry> FuncInfoArray;

}

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

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

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

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

#include "RuntimeByteCodePch.h"

#include "BackEndOpcodeAttr.h"

namespace OpCodeAttr

{

// OpSideEffect:

// Opcode has side effect not just to the dst/src on the instruction.

// The opcode cannot be deadstored. (e.g. StFld, LdFld from DOM, call valueOf/toString/getter/setter)

// Doesn't include all "exit" script (e.g. LdThis doesn't have side effect for HostDispatch for exiting script to getting the name space parent)

// OpHasImplicitCall:

// Include all possible exit scripts, call valueOf/toString/getter/setter

// OpSerialized:

// Op is a serialized (indirected) variant of another op code

enum OpCodeAttrEnum

{

None = 0x00000000,

OpSideEffect = 0x00000001, // If dst is unused and src can’t call implicitcalls, it still can not be dead-stored (Could throw an exception, etc)

OpUseAllFields = 0x00000002,

OpTempNumberSources = 0x00000004, // OpCode does support temp values as source

OpTempNumberProducing = 0x00000008, // OpCode can produce a temp value

OpTempNumberTransfer = 0x00000010 | OpTempNumberSources, // OpCode transfers a temp value

OpTempObjectSources = 0x00000020, // OpCode does support temp values as source

OpTempObjectProducing = 0x00000040, // OpCode can produce a temp value

OpTempObjectTransfer = 0x00000080 | OpTempObjectSources, // OpCode transfers a temp value

OpTempObjectCanStoreTemp = 0x00000100 | OpTempObjectProducing, // OpCode can produce a temp value, and once marked, it will always produce a temp value so we can store other temp value in the object

OpInlineCallInstr = 0x00000200,

OpCallsValueOf = 0x00000400, // Could have valueOf/ToString side-effect

OpCallInstr = 0x00000800,

OpDoNotTransfer = 0x00001000,

OpHasImplicitCall = 0x00002000, // Evaluating the src may call JS user code implicitly (getters/setters/valueof/tostring/DOM callbacks/etc)

OpFastFldInstr = 0x00004000,

OpBailOutRec = 0x00008000,

OpInlinableBuiltIn = 0x00010000, // OpCode is an inlineable built-in, such as InlineMathSin, etc.

OpNonIntTransfer = 0x00020000, // OpCode may transfer a non-integer value from the non-constant source to the destination

OpIsInt32 = 0x00040000, // OpCode converts its srcs to int32 or a narrower int type, and produces an int32

OpProducesNumber = 0x00080000, // OpCode always produces a number

OpCanLoadFixedFields = 0x00100000, // OpCode can use fixed fields

OpCanCSE = 0x00200000, // Opcode has no side-effect and always produces the same value for a given input (InlineMathAbs is OK, InlineMathRandom is not)

OpNoFallThrough = 0x00400000, // Opcode doesn't fallthrough in flow and it always jumps to the return from this opcode

OpPostOpDbgBailOut = 0x00800000, // Generate bail out after this opcode. This must be a helper call and needs bailout on return from it. Used for Fast F12.

OpHasMultiSizeLayout = 0x01000000,

OpHasProfiled = 0x02000000,

OpHasProfiledWithICIndex = 0x04000000,

OpDeadFallThrough = 0x08000000,

OpProfiled = 0x10000000, // OpCode is a profiled variant

OpProfiledWithICIndex = 0x20000000, // OpCode is a profiled with IC index variant

OpBackEndOnly = 0x40000000,

OpByteCodeOnly = 0x80000000,

};

static const int OpcodeAttributes[] =

{

#define DEF\_OP(name, jnLayout, attrib, ...) attrib,

#include "ByteCode\OpCodeList.h"

#undef DEF\_OP

};

static const int ExtendedOpcodeAttributes[] =

{

#define DEF\_OP(name, jnLayout, attrib, ...) attrib,

#include "ByteCode\ExtendedOpCodeList.h"

#undef DEF\_OP

};

static const int BackendOpCodeAttributes[] =

{

#define DEF\_OP(name, jnLayout, attrib, ...) attrib,

#include "BackendOpCodeList.h"

#undef DEF\_OP

};

static const int GetOpCodeAttributes(Js::OpCode op)

{

if (op <= Js::OpCode::MaxByteSizedOpcodes)

{

AnalysisAssert(op < \_countof(OpcodeAttributes));

return OpcodeAttributes[(int)op];

}

else if (op < Js::OpCode::ByteCodeLast)

{

uint opIndex = op - (Js::OpCode::MaxByteSizedOpcodes + 1);

AnalysisAssert(opIndex < \_countof(ExtendedOpcodeAttributes));

return ExtendedOpcodeAttributes[opIndex];

}

uint opIndex = op - (Js::OpCode::ByteCodeLast + 1);

AnalysisAssert(opIndex < \_countof(BackendOpCodeAttributes));

return BackendOpCodeAttributes[opIndex];

}

bool HasSideEffects(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpSideEffect) != 0);

};

bool CanCSE(Js::OpCode opcode)

{

Assert(((GetOpCodeAttributes(opcode) & OpCanCSE) == 0) || ((GetOpCodeAttributes(opcode) & OpSideEffect) == 0));

return ((GetOpCodeAttributes(opcode) & OpCanCSE) != 0);

};

bool UseAllFields(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpUseAllFields) != 0);

}

bool NonTempNumberSources(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpTempNumberSources) == 0);

}

bool TempNumberSources(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpTempNumberSources) != 0);

}

bool TempNumberProducing(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpTempNumberProducing) != 0);

}

bool TempNumberTransfer(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpTempNumberTransfer) == OpTempNumberTransfer);

}

bool TempObjectSources(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpTempObjectSources) != 0);

}

bool TempObjectProducing(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpTempObjectProducing) != 0);

}

bool TempObjectTransfer(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpTempObjectTransfer) == OpTempObjectTransfer);

}

bool TempObjectCanStoreTemp(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpTempObjectCanStoreTemp) == OpTempObjectCanStoreTemp);

}

bool CallInstr(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpCallInstr) != 0);

}

bool InlineCallInstr(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpInlineCallInstr) != 0);

}

bool CallsValueOf(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpCallsValueOf) != 0);

}

bool FastFldInstr(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpFastFldInstr) != 0);

}

bool BailOutRec(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpBailOutRec) != 0);

}

bool ByteCodeOnly(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpByteCodeOnly) != 0);

}

bool BackEndOnly(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpBackEndOnly) != 0);

}

bool DoNotTransfer(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpDoNotTransfer) != 0);

}

bool HasImplicitCall(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpHasImplicitCall) != 0);

}

bool IsProfiledOp(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpProfiled) != 0);

}

bool IsProfiledOpWithICIndex(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpProfiledWithICIndex) != 0);

}

bool IsInlineBuiltIn(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpInlinableBuiltIn) != 0);

}

bool NonIntTransfer(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpNonIntTransfer) != 0);

}

bool IsInt32(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpIsInt32) != 0);

}

bool ProducesNumber(Js::OpCode opcode, Js::ScriptContext \*const scriptContext)

{

return ((GetOpCodeAttributes(opcode) & OpProducesNumber) != 0);

}

bool HasFallThrough(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpNoFallThrough) == 0);

}

bool NeedsPostOpDbgBailOut(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpPostOpDbgBailOut) != 0);

}

bool HasMultiSizeLayout(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpHasMultiSizeLayout) != 0);

}

bool HasProfiledOp(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpHasProfiled) != 0);

}

bool HasProfiledOpWithICIndex(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpHasProfiledWithICIndex) != 0);

}

bool HasDeadFallThrough(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpDeadFallThrough) != 0);

}

bool CanLoadFixedFields(Js::OpCode opcode)

{

return ((GetOpCodeAttributes(opcode) & OpCanLoadFixedFields) != 0);

}

}; // OpCodeAttr

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

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

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

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

namespace OpCodeAttr

{

// True if the opcode has side effect other then assigning to the dst

bool HasSideEffects(Js::OpCode opcode);

// True if the opcode has not side-effect and always produces the same value for a given input

bool CanCSE(Js::OpCode opcode);

// True if the opcode implicitly (may) use any the fields

bool UseAllFields(Js::OpCode opcode);

// True if the opcode don't allow temp numbers sources (e.g. StFld, Ret, StElem)

bool NonTempNumberSources(Js::OpCode opcode);

// !NonTempSources

bool TempNumberSources(Js::OpCode opcode);

// True if the opcode can use a temporary stack slot for it's number result

bool TempNumberProducing(Js::OpCode opcode);

// True if the source number may assign directly to the dest number

bool TempNumberTransfer(Js::OpCode opcode);

// True if the opcode allows temp object sources

bool TempObjectSources(Js::OpCode opcode);

// True if the opcode can generate object on the stack

bool TempObjectProducing(Js::OpCode opcode);

// True if the opcode can generate object on the stack and will once marked so other temp object/number can be stored

bool TempObjectCanStoreTemp(Js::OpCode opcode);

// True if the source object may assign directly to the dest object

bool TempObjectTransfer(Js::OpCode opcode);

// True for call instructions

bool CallInstr(Js::OpCode opcode);

// True for call instructions which may get inlined

bool InlineCallInstr(Js::OpCode opcode);

// True if the opcode may have the side-effect of calling valueof().

bool CallsValueOf(Js::OpCode opcode);

// True if the opcode can do optimizations on property syms.

bool FastFldInstr(Js::OpCode opcode);

// True if the opcode requires a bailout record.

bool BailOutRec(Js::OpCode opcode);

// True if the opcode can only appear in the byte code

bool ByteCodeOnly(Js::OpCode opcode);

// True if the opcode can only appear in the back end

bool BackEndOnly(Js::OpCode opcode);

// True if the opcode does not transfer value from src to dst (only some opcode are marked property)

bool DoNotTransfer(Js::OpCode opcode);

// True if the opcode may have implicit call

bool HasImplicitCall(Js::OpCode opcode);

// True if the opcode is a profiled variant

bool IsProfiledOp(Js::OpCode opcode);

// True if the opcode is a profiled variant with an inline cache index

bool IsProfiledOpWithICIndex(Js::OpCode opcode);

// True if the opcode is a math helper, such as sin, cos, pow,..

bool IsInlineBuiltIn(Js::OpCode opcode);

// True if the opcode may transfer a non-integer value from the non-constant source to the destination

bool NonIntTransfer(Js::OpCode opcode);

// True if the opcode converts its srcs to int32 or a narrower int type, and produces an int32

bool IsInt32(Js::OpCode opcode);

// True if the opcode always produces a number in the specified script context's version

bool ProducesNumber(Js::OpCode opcode, Js::ScriptContext \*const scriptContext);

// False if the opcode results in jump to end of the function and there cannot be fallthrough.

bool HasFallThrough(Js::OpCode opcode);

// True if we need to generate bailout after this opcode when in debug mode (b/o on return from helper).

bool NeedsPostOpDbgBailOut(Js::OpCode opcode);

// True if the opcode has a small/large layout

bool HasMultiSizeLayout(Js::OpCode opcode);

// True if the opcode has a profiled version of the opcode

bool HasProfiledOp(Js::OpCode opcode);

// True if the opcode has a profiled version of the opcode with an inline cache index

bool HasProfiledOpWithICIndex(Js::OpCode opcode);

// True if the opcode will never fallthrough (e.g. guaranteed bailout)

bool HasDeadFallThrough(Js::OpCode opcode);

// True if the opcode can use fixed fields

bool CanLoadFixedFields(Js::OpCode opcode);

};

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

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

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

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

#include "RuntimeByteCodePch.h"

namespace Js

{

uint ByteBlock::GetLength() const

{

return m\_contentSize;

}

const byte\* ByteBlock::GetBuffer() const

{

return m\_content;

}

byte\* ByteBlock::GetBuffer()

{

return m\_content;

}

const byte ByteBlock::operator[](uint itemIndex) const

{

AssertMsg(itemIndex < m\_contentSize, "Ensure valid offset");

return m\_content[itemIndex];

}

byte& ByteBlock::operator[] (uint itemIndex)

{

AssertMsg(itemIndex < m\_contentSize, "Ensure valid offset");

return m\_content[itemIndex];

}

ByteBlock \*ByteBlock::New(Recycler \*alloc, const byte \* initialContent, int initialContentSize)

{

// initialContent may be 'null' if no data to copy

AssertMsg(initialContentSize > 0, "Must have valid data size");

ByteBlock \*newBlock = RecyclerNew(alloc, ByteBlock, initialContentSize, alloc);

//

// Copy any optional data into the block:

// - If initialContent was not provided, the block's contents will be uninitialized.

//

if (initialContent != nullptr)

{

js\_memcpy\_s(newBlock->m\_content, newBlock->GetLength(), initialContent, initialContentSize);

}

return newBlock;

}

ByteBlock \*ByteBlock::NewFromArena(ArenaAllocator \*alloc, const byte \* initialContent, int initialContentSize)

{

// initialContent may be 'null' if no data to copy

AssertMsg(initialContentSize > 0, "Must have valid data size");

ByteBlock \*newBlock = Anew(alloc, ByteBlock, initialContentSize, alloc);

//

// Copy any optional data into the block:

// - If initialContent was not provided, the block's contents will be uninitialized.

//

if (initialContent != nullptr)

{

js\_memcpy\_s(newBlock->m\_content, newBlock->GetLength(), initialContent, initialContentSize);

}

return newBlock;

}

ByteBlock \* ByteBlock::Clone(Recycler\* alloc)

{

return ByteBlock::New(alloc, this->m\_content, this->m\_contentSize);

}

ByteBlock \*ByteBlock::New(Recycler \*alloc, const byte \* initialContent, int initialContentSize, ScriptContext \* requestContext)

{

// initialContent may be 'null' if no data to copy

AssertMsg(initialContentSize > 0, "Must have valid data size");

ByteBlock \*newBlock = RecyclerNew(alloc, ByteBlock, initialContentSize, alloc);

//

// Copy any optional data into the block:

// - If initialContent was not provided, the block's contents will be uninitialized.

//

if (initialContent != nullptr)

{

//

// Treat initialContent as array of vars

// Clone vars to the requestContext

//

Var \*src = (Var\*)initialContent;

Var \*dst = (Var\*)newBlock->m\_content;

size\_t count = initialContentSize / sizeof(Var);

for (size\_t i = 0; i < count; i++)

{

if (TaggedInt::Is(src[i]))

{

dst[i] = src[i];

}

else

{

//

// Currently only numbers are put into AuxiliaryContext data

//

Assert(JavascriptNumber::Is(src[i]));

dst[i] = JavascriptNumber::CloneToScriptContext(src[i], requestContext);

requestContext->BindReference(dst[i]);

}

}

}

return newBlock;

}

//

// Create a copy of buffer

// Each Var is cloned on the requestContext

//

ByteBlock \* ByteBlock::Clone(Recycler\* alloc, ScriptContext \* requestContext)

{

return ByteBlock::New(alloc, this->m\_content, this->m\_contentSize, requestContext);

}

}

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

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

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

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

#pragma once

namespace Js

{

class ByteBlock

{

DECLARE\_OBJECT(ByteBlock)

private:

uint m\_contentSize; // Length of block, in bytes

\_\_declspec(align(4)) // Align the buffer to sizeof(uint32) to improve GetHashCode() perf.

byte\* m\_content; // The block's content

static ByteBlock\* New(Recycler\* alloc, const byte \* initialContent, int initialContentSize, ScriptContext \* requestContext);

public:

ByteBlock(uint size, byte \* content)

: m\_contentSize(size), m\_content(content)

{ }

ByteBlock(uint size, Recycler \*alloc) : m\_contentSize(size)

{

// The New function below will copy over a buffer into this so

// we don't need to zero it out

m\_content = RecyclerNewArrayLeaf(alloc, byte, size);

}

ByteBlock(uint size, ArenaAllocator\* alloc) : m\_contentSize(size)

{

m\_content = AnewArray(alloc, byte, size);

}

static DWORD GetBufferOffset() { return offsetof(ByteBlock, m\_content); }

static ByteBlock\* New(Recycler\* alloc, const byte \* initialContent, int initialContentSize);

// This is needed just for the debugger since it allocates

// the byte block on a separate thread, which the recycler doesn't like.

// To remove when the recycler supports multi-threaded allocation.

static ByteBlock\* NewFromArena(ArenaAllocator\* alloc, const byte \* initialContent, int initialContentSize);

uint GetLength() const;

byte\* GetBuffer();

const byte\* GetBuffer() const;

const byte operator[](uint itemIndex) const;

byte& operator[] (uint itemIndex);

ByteBlock \* Clone(Recycler\* alloc);

//

// Create a copy of buffer

// Each Var is cloned on the requestContext

//

ByteBlock \* Clone(Recycler\* alloc, ScriptContext \* requestContext);

};

} // 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.

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

#pragma once

HRESULT GenerateByteCode(\_\_in ParseNode \*pnode, \_\_in ulong grfscr, \_\_in Js::ScriptContext\* scriptContext, \_\_inout Js::ParseableFunctionInfo \*\* ppRootFunc,

\_\_in uint sourceIndex, \_\_in bool forceNoNative, \_\_in Parser\* parser, \_\_in CompileScriptException \*pse, Js::ScopeInfo\* parentScopeInfo = nullptr,

Js::ScriptFunction \*\* functionRef = nullptr);

//

// Output for -Trace:ByteCode

//

#if DBG\_DUMP

#define TRACE\_BYTECODE(fmt, ...) \

if (Js::Configuration::Global.flags.Trace.IsEnabled(Js::ByteCodePhase)) \

{ \

Output::Print(fmt, \_\_VA\_ARGS\_\_); \

}

#else

#define TRACE\_BYTECODE(fmt, ...)

#endif

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

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

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

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

// NOTE: If there is a merge conflict the correct fix is to make a new GUID.

// {A1B09088-45E7-42DB-8D65-FB739F559946}

const GUID byteCodeCacheReleaseFileVersion =

{ 0xa1b09088, 0x45e7, 0x42db,{ 0x8d, 0x65, 0xfb, 0x73, 0x9f, 0x55, 0x99, 0x46 } };

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

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

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

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

#include "RuntimeByteCodePch.h"

#if DBG\_DUMP

#if DBG

// Parser Includes

#include "RegexCommon.h"

#include "DebugWriter.h"

#include "RegexPattern.h"

#endif

namespace Js

{

// Pre-order recursive dump, head first, then children.

void ByteCodeDumper::DumpRecursively(FunctionBody\* dumpFunction)

{

dumpFunction->EnsureDeserialized();

ByteCodeDumper::Dump(dumpFunction);

for (uint i = 0; i < dumpFunction->GetNestedCount(); i ++)

{

dumpFunction->GetNestedFunc(i)->EnsureDeserialized();

ByteCodeDumper::DumpRecursively(dumpFunction->GetNestedFunc(i)->GetFunctionBody());

}

}

void ByteCodeDumper::Dump(FunctionBody\* dumpFunction)

{

if (!CONFIG\_FLAG(DumpDbgControllerBytecode) && dumpFunction->GetSourceContextInfo() &&

dumpFunction->GetSourceContextInfo()->url != nullptr &&

\_wcsicmp(dumpFunction->GetSourceContextInfo()->url, L"dbgcontroller.js") == 0)

{

return;

}

ByteCodeReader reader;

reader.Create(dumpFunction);

StatementReader statementReader;

statementReader.Create(dumpFunction);

dumpFunction->DumpFullFunctionName();

Output::Print(L" (");

ArgSlot inParamCount = dumpFunction->GetInParamsCount();

for (ArgSlot paramIndex = 0; paramIndex < inParamCount; paramIndex++)

{

if (paramIndex > 0)

{

Output::Print(L", ");

}

Output::Print(L"In%hu", paramIndex);

}

Output::Print(L") ");

Output::Print(L"(size: %d [%d])\n", dumpFunction->GetByteCodeCount(), dumpFunction->GetByteCodeWithoutLDACount());

#if defined(DBG) || defined(ENABLE\_DEBUG\_CONFIG\_OPTIONS)

if (dumpFunction->IsByteCodeDebugMode())

{

Output::Print(L"[Bytecode was generated for debug mode]\n");

}

#endif

#if DBG

if (dumpFunction->IsReparsed())

{

Output::Print(L"[A reparse is being done]\n");

}

#endif

Output::Print(

L" %u locals (%u temps from R%u), %u inline cache\n",

dumpFunction->GetLocalsCount(),

dumpFunction->GetTempCount(),

dumpFunction->GetFirstTmpReg(),

dumpFunction->GetInlineCacheCount());

uint32 statementIndex = 0;

ByteCodeDumper::DumpConstantTable(dumpFunction);

ByteCodeDumper::DumpImplicitArgIns(dumpFunction);

while (true)

{

while (statementReader.AtStatementBoundary(&reader))

{

dumpFunction->PrintStatementSourceLine(statementIndex);

statementIndex = statementReader.MoveNextStatementBoundary();

}

uint byteOffset = reader.GetCurrentOffset();

LayoutSize layoutSize;

OpCode op = reader.ReadOp(layoutSize);

if (op == OpCode::EndOfBlock)

{

Assert(reader.GetCurrentOffset() == dumpFunction->GetByteCode()->GetLength());

break;

}

Output::Print(L" %04x %2s", byteOffset, layoutSize == LargeLayout? L"L-" : layoutSize == MediumLayout? L"M-" : L"");

DumpOp(op, layoutSize, reader, dumpFunction);

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

{

int layoutStart = byteOffset + 2; // Account fo the prefix op

int endByteOffset = reader.GetCurrentOffset();

Output::SkipToColumn(70);

if (layoutSize == LargeLayout)

{

Output::Print(L"%02X ",

op > Js::OpCode::MaxByteSizedOpcodes?

Js::OpCode::ExtendedLargeLayoutPrefix : Js::OpCode::LargeLayoutPrefix);

}

else if (layoutSize == MediumLayout)

{

Output::Print(L"%02X ",

op > Js::OpCode::MaxByteSizedOpcodes?

Js::OpCode::ExtendedMediumLayoutPrefix : Js::OpCode::MediumLayoutPrefix);

}

else

{

Assert(layoutSize == SmallLayout);

if (op > Js::OpCode::MaxByteSizedOpcodes)

{

Output::Print(L"%02X ", Js::OpCode::ExtendedOpcodePrefix);

}

else

{

Output::Print(L" ");

layoutStart--; // don't have a prefix

}

}

Output::Print(L"%02x", (byte)op);

for (int i = layoutStart; i < endByteOffset; i++)

{

Output::Print(L" %02x", reader.GetRawByte(i));

}

}

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

}

if (statementReader.AtStatementBoundary(&reader))

{

dumpFunction->PrintStatementSourceLine(statementIndex);

statementIndex = statementReader.MoveNextStatementBoundary();

}

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

Output::Flush();

}

void ByteCodeDumper::DumpConstantTable(FunctionBody \*dumpFunction)

{

Output::Print(L" Constant Table:\n ======== =====\n ");

uint count = dumpFunction->GetConstantCount();

for (RegSlot reg = FunctionBody::FirstRegSlot; reg < count; reg++)

{

DumpReg(reg);

Var varConst = dumpFunction->GetConstantVar(reg);

Assert(varConst != nullptr);

if (TaggedInt::Is(varConst))

{

#if ENABLE\_NATIVE\_CODEGEN

Output::Print(L"%-10s", OpCodeUtil::GetOpCodeName(OpCode::LdC\_A\_I4));

#else

Output::Print(L"%-10s", OpCodeUtil::GetOpCodeName(OpCode::Ld\_A));

#endif

DumpI4(TaggedInt::ToInt32(varConst));

}

else if (varConst == (Js::Var)&Js::NullFrameDisplay)

{

#if ENABLE\_NATIVE\_CODEGEN

Output::Print(L"%-10s", OpCodeUtil::GetOpCodeName(OpCode::LdNullDisplay));

#else

Output::Print(L"%-10s", OpCodeUtil::GetOpCodeName(OpCode::Ld\_A));

Output::Print(L" (NullDisplay)");

#endif

}

else if (varConst == (Js::Var)&Js::StrictNullFrameDisplay)

{

#if ENABLE\_NATIVE\_CODEGEN

Output::Print(L"%-10s", OpCodeUtil::GetOpCodeName(OpCode::LdStrictNullDisplay));

#else

Output::Print(L"%-10s", OpCodeUtil::GetOpCodeName(OpCode::Ld\_A));

Output::Print(L" (StrictNullDisplay)");

#endif

}

else

{

switch (JavascriptOperators::GetTypeId(varConst))

{

case Js::TypeIds\_Undefined:

Output::Print(L"%-10s", OpCodeUtil::GetOpCodeName(OpCode::Ld\_A));

Output::Print(L" (undefined)");

break;

case Js::TypeIds\_Null:

Output::Print(L"%-10s", OpCodeUtil::GetOpCodeName(OpCode::Ld\_A));

Output::Print(L" (null)");

break;

case Js::TypeIds\_Boolean:

Output::Print(L"%-10s", OpCodeUtil::GetOpCodeName(

JavascriptBoolean::FromVar(varConst)->GetValue() ? OpCode::LdTrue : OpCode::LdFalse));

break;

case Js::TypeIds\_Number:

#if ENABLE\_NATIVE\_CODEGEN

Output::Print(L"%-10s", OpCodeUtil::GetOpCodeName(OpCode::LdC\_A\_R8));

#else

Output::Print(L"%-10s", OpCodeUtil::GetOpCodeName(OpCode::Ld\_A));

#endif

Output::Print(L"%G", JavascriptNumber::GetValue(varConst));

break;

case Js::TypeIds\_String:

#if ENABLE\_NATIVE\_CODEGEN

Output::Print(L"%-10s", OpCodeUtil::GetOpCodeName(OpCode::LdStr));

#else

Output::Print(L"%-10s", OpCodeUtil::GetOpCodeName(OpCode::Ld\_A));

#endif

Output::Print(L" (\"%s\")", JavascriptString::FromVar(varConst)->GetSz());

break;

case Js::TypeIds\_GlobalObject:

#if ENABLE\_NATIVE\_CODEGEN

Output::Print(L"%-10s", OpCodeUtil::GetOpCodeName(OpCode::LdRoot));

#else

Output::Print(L"%-10s", OpCodeUtil::GetOpCodeName(OpCode::Ld\_A));

#endif

break;

case Js::TypeIds\_ModuleRoot:

#if ENABLE\_NATIVE\_CODEGEN

Output::Print(L"%-10s", OpCodeUtil::GetOpCodeName(OpCode::LdModuleRoot));

#else

Output::Print(L"%-10s", OpCodeUtil::GetOpCodeName(OpCode::Ld\_A));

#endif

DumpI4(dumpFunction->GetModuleID());

break;

case Js::TypeIds\_ES5Array:

// ES5Array objects in the constant table are always string template callsite objects.

// If we later put other ES5Array objects in the constant table, we'll need another way

// to decide the constant type.

Output::Print(L"%-10s", L"LdStringTemplate");

Output::Print(L" (\"%s\")", dumpFunction->GetScriptContext()->GetLibrary()->GetStringTemplateCallsiteObjectKey(varConst));

break;

default:

AssertMsg(UNREACHED, "Unexpected object type in CloneConstantTable");

break;

}

}

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

}

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

}

void ByteCodeDumper::DumpImplicitArgIns(FunctionBody \* dumpFunction)

{

if (dumpFunction->GetInParamsCount() <= 1 || !dumpFunction->GetHasImplicitArgIns())

{

return;

}

Output::Print(L" Implicit Arg Ins:\n ======== === ===\n ");

for (RegSlot reg = 1;

reg < dumpFunction->GetInParamsCount(); reg++)

{

DumpReg((RegSlot)(reg + dumpFunction->GetConstantCount() - 1));

// DisableJIT-TODO: Should this entire function be ifdefed?

#if ENABLE\_NATIVE\_CODEGEN

Output::Print(L"%-11s", OpCodeUtil::GetOpCodeName(Js::OpCode::ArgIn\_A));

#endif

Output::Print(L"In%d\n ", reg);

}

if (dumpFunction->GetHasRestParameter())

{

DumpReg(dumpFunction->GetRestParamRegSlot());

#if ENABLE\_NATIVE\_CODEGEN

Output::Print(L"%-11s", OpCodeUtil::GetOpCodeName(Js::OpCode::ArgIn\_Rest));

#endif

Output::Print(L"In%d\n ", dumpFunction->GetInParamsCount());

}

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

}

void ByteCodeDumper::DumpU4(uint32 value)

{

Output::Print(L" uint:%u ", value);

}

void ByteCodeDumper::DumpI4(int value)

{

Output::Print(L" int:%d ", value);

}

void ByteCodeDumper::DumpU2(ushort value)

{

Output::Print(L" ushort:%d ", value);

}

void ByteCodeDumper::DumpOffset(int byteOffset, ByteCodeReader const& reader)

{

Output::Print(L" x:%04x (%4d) ", reader.GetCurrentOffset() + byteOffset, byteOffset);

}

void ByteCodeDumper::DumpAddr(void\* addr)

{

Output::Print(L" addr:%04x ", addr);

}

void ByteCodeDumper::DumpOp(OpCode op, LayoutSize layoutSize, ByteCodeReader& reader, FunctionBody\* dumpFunction)

{

Output::Print(L"%-20s", OpCodeUtil::GetOpCodeName(op));

OpLayoutType nType = OpCodeUtil::GetOpCodeLayout(op);

switch (layoutSize \* OpLayoutType::Count + nType)

{

#define LAYOUT\_TYPE(layout) \

case OpLayoutType::layout: \

Assert(layoutSize == SmallLayout); \

Dump##layout(op, reader.layout(), dumpFunction, reader); \

break;

#define LAYOUT\_TYPE\_WMS(layout) \

case SmallLayout \* OpLayoutType::Count + OpLayoutType::layout: \

Dump##layout(op, reader.layout##\_Small(), dumpFunction, reader); \

break; \

case MediumLayout \* OpLayoutType::Count + OpLayoutType::layout: \

Dump##layout(op, reader.layout##\_Medium(), dumpFunction, reader); \

break; \

case LargeLayout \* OpLayoutType::Count + OpLayoutType::layout: \

Dump##layout(op, reader.layout##\_Large(), dumpFunction, reader); \

break;

#include "LayoutTypes.h"

default:

{

AssertMsg(false, "Unknown OpLayout");

break;

}

}

}

void ByteCodeDumper::DumpR8(double value)

{

Output::Print(L" double:%g ", value);

}

void ByteCodeDumper::DumpReg(RegSlot registerID)

{

Output::Print(L" R%d ", (int) registerID);

}

void ByteCodeDumper::DumpReg(RegSlot\_TwoByte registerID)

{

Output::Print(L" R%d ", (int) registerID);

}

void ByteCodeDumper::DumpReg(RegSlot\_OneByte registerID)

{

Output::Print(L" R%d ", (int) registerID);

}

void ByteCodeDumper::DumpProfileId(uint id)

{

Output::Print(L" <%d> ", id);

}

void ByteCodeDumper::DumpEmpty(OpCode op, const unaligned OpLayoutEmpty \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

}

template <class T>

void ByteCodeDumper::DumpCallI(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

if (data->Return != Constants::NoRegister)

{

DumpReg((RegSlot)data->Return);

Output::Print(L"=");

}

Output::Print(L" R%d(ArgCount: %d)", data->Function, data->ArgCount);

}

template <class T>

void ByteCodeDumper::DumpCallIExtended(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpCallI(op, data, dumpFunction, reader);

if (data->Options & Js::CallIExtended\_SpreadArgs)

{

const Js::AuxArray<uint32> \*arr = reader.ReadAuxArray<uint32>(data->SpreadAuxOffset, dumpFunction);

Output::Print(L" spreadArgs [", arr->count);

for (uint i = 0; i < arr->count; i++)

{

if (i > 10)

{

Output::Print(L", ...");

break;

}

if (i != 0)

{

Output::Print(L", ");

}

Output::Print(L"%u", arr->elements[i]);

}

Output::Print(L"]");

}

}

template <class T>

void ByteCodeDumper::DumpCallIFlags(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpCallI(op, data, dumpFunction, reader);

Output::Print(L" <%04x> ", data->callFlags);

}

template <class T>

void ByteCodeDumper::DumpCallIExtendedFlags(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpCallIFlags(op, data, dumpFunction, reader);

if (data->Options & Js::CallIExtended\_SpreadArgs)

{

const Js::AuxArray<uint32> \*arr = reader.ReadAuxArray<uint32>(data->SpreadAuxOffset, dumpFunction);

Output::Print(L" spreadArgs [", arr->count);

for (uint i = 0; i < arr->count; i++)

{

if (i > 10)

{

Output::Print(L", ...");

break;

}

if (i != 0)

{

Output::Print(L", ");

}

Output::Print(L"%u", arr->elements[i]);

}

Output::Print(L"]");

}

}

template <class T>

void ByteCodeDumper::DumpCallIExtendedFlagsWithICIndex(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpCallIFlags(op, data, dumpFunction, reader);

DumpCallIWithICIndex(op, data, dumpFunction, reader);

if (data->Options & Js::CallIExtended\_SpreadArgs)

{

const Js::AuxArray<uint32> \*arr = reader.ReadAuxArray<uint32>(data->SpreadAuxOffset, dumpFunction);

Output::Print(L" spreadArgs [", arr->count);

for (uint i = 0; i < arr->count; i++)

{

if (i > 10)

{

Output::Print(L", ...");

break;

}

if (i != 0)

{

Output::Print(L", ");

}

Output::Print(L"%u", arr->elements[i]);

}

Output::Print(L"]");

}

}

template <class T>

void ByteCodeDumper::DumpCallIWithICIndex(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpCallI(op, data, dumpFunction, reader);

Output::Print(L" <%d> ", data->inlineCacheIndex);

}

template <class T>

void ByteCodeDumper::DumpCallIFlagsWithICIndex(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpCallI(op, data, dumpFunction, reader);

Output::Print(L" <%d> ", data->inlineCacheIndex);

Output::Print(L" <%d> ", data->callFlags);

}

template <class T>

void ByteCodeDumper::DumpCallIExtendedWithICIndex(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpCallIWithICIndex(op, data, dumpFunction, reader);

if (data->Options & Js::CallIExtended\_SpreadArgs)

{

const Js::AuxArray<uint32> \*arr = reader.ReadAuxArray<uint32>(data->SpreadAuxOffset, dumpFunction);

Output::Print(L" spreadArgs [", arr->count);

for (uint i=0; i < arr->count; i++)

{

if (i > 10)

{

Output::Print(L", ...");

break;

}

if (i != 0)

{

Output::Print(L", ");

}

Output::Print(L"%u", arr->elements[i]);

}

Output::Print(L"]");

}

}

template <class T>

void ByteCodeDumper::DumpElementI(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

switch (op)

{

case OpCode::ProfiledLdElemI\_A:

case OpCode::LdElemI\_A:

case OpCode::LdMethodElem:

case OpCode::TypeofElem:

{

Output::Print(L" R%d = R%d[R%d]", data->Value, data->Instance, data->Element);

break;

}

case OpCode::ProfiledStElemI\_A:

case OpCode::ProfiledStElemI\_A\_Strict:

case OpCode::StElemI\_A:

case OpCode::StElemI\_A\_Strict:

case OpCode::InitSetElemI:

case OpCode::InitGetElemI:

case OpCode::InitComputedProperty:

case OpCode::InitClassMemberComputedName:

case OpCode::InitClassMemberGetComputedName:

case OpCode::InitClassMemberSetComputedName:

{

Output::Print(L" R%d[R%d] = R%d", data->Instance, data->Element, data->Value);

break;

}

case OpCode::DeleteElemI\_A:

case OpCode::DeleteElemIStrict\_A:

{

Output::Print(L" R%d[R%d]", data->Instance, data->Element);

break;

}

default:

{

AssertMsg(false, "Unknown OpCode for OpLayoutElementI");

break;

}

}

}

template <class T>

void ByteCodeDumper::DumpReg2Int1(OpCode op, const unaligned T\* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

switch (op)

{

case OpCode::LdThis:

case OpCode::ProfiledLdThis:

Output::Print(L" R%d = R%d, %d", data->R0, data->R1, data->C1);

break;

case OpCode::LdIndexedFrameDisplay:

Output::Print(L" R%d = [%d], R%d ", data->R0, data->C1, data->R1);

break;

case OpCode::GetCachedFunc:

DumpReg(data->R0);

Output::Print(L"= func(");

DumpReg(data->R1);

Output::Print(L",");

DumpI4(data->C1);

Output::Print(L")");

break;

default:

AssertMsg(false, "Unknown OpCode for OpLayoutReg2Int1");

break;

}

}

template <class T>

void ByteCodeDumper::DumpElementScopedU(OpCode op, const unaligned T \* data, Js::FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

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

PropertyRecord const \* pPropertyName = scriptContext->GetPropertyName(

dumpFunction->GetReferencedPropertyId(data->PropertyIdIndex));

switch (op)

{

case OpCode::LdElemUndefScoped:

{

Output::Print(L" %s = undefined, R%d", pPropertyName->GetBuffer(), Js::FunctionBody::RootObjectRegSlot);

break;

}

case OpCode::InitUndeclConsoleLetFld:

case OpCode::InitUndeclConsoleConstFld:

{

Output::Print(L" %s = undefined", pPropertyName->GetBuffer());

break;

}

default:

{

AssertMsg(false, "Unknown OpCode for ElementScopedU");

break;

}

}

}

template <class T>

void ByteCodeDumper::DumpElementU(OpCode op, const unaligned T \* data, Js::FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

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

PropertyRecord const \* pPropertyName = scriptContext->GetPropertyName(

dumpFunction->GetReferencedPropertyId(data->PropertyIdIndex));

switch (op)

{

case OpCode::LdElemUndef:

{

Output::Print(L" R%d.%s = undefined", data->Instance, pPropertyName->GetBuffer());

break;

}

// TODO: Change InitUndeclLetFld and InitUndeclConstFld to ElementU layout

// case OpCode::InitUndeclLetFld:

// case OpCode::InitUndeclConstFld:

// {

// PropertyRecord const \* pPropertyName = scriptContext->GetPropertyName(data->PropertyIndex);

// Output::Print(L" R%d.%s", data->Instance, pPropertyName->GetBuffer());

// break;

// }

case OpCode::ClearAttributes:

{

Output::Print(L" R%d.%s.writable/enumerable/configurable = 0", data->Instance, pPropertyName->GetBuffer());

break;

}

case OpCode::DeleteLocalFld:

Output::Print(L" R%d = %s ", data->Instance, pPropertyName->GetBuffer());

break;

case OpCode::StLocalFuncExpr:

Output::Print(L" %s = R%d", pPropertyName->GetBuffer(), data->Instance);

break;

default:

{

AssertMsg(false, "Unknown OpCode for ElementU");

break;

}

}

}

template <class T>

void ByteCodeDumper::DumpElementRootU(OpCode op, const unaligned T \* data, Js::FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

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

PropertyRecord const \* pPropertyName = scriptContext->GetPropertyName(

dumpFunction->GetReferencedPropertyId(data->PropertyIdIndex));

switch (op)

{

case OpCode::InitUndeclRootLetFld:

case OpCode::InitUndeclRootConstFld:

case OpCode::EnsureNoRootFld:

case OpCode::EnsureNoRootRedeclFld:

{

Output::Print(L" root.%s", pPropertyName->GetBuffer());

break;

}

case OpCode::LdLocalElemUndef:

{

Output::Print(L" %s = undefined", pPropertyName->GetBuffer());

break;

}

default:

{

AssertMsg(false, "Unknown OpCode for ElementRootU");

break;

}

}

}

template <class T>

void ByteCodeDumper::DumpElementScopedC(OpCode op, const unaligned T \* data, Js::FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

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

PropertyRecord const \* pPropertyName = scriptContext->GetPropertyName(

dumpFunction->GetReferencedPropertyId(data->PropertyIdIndex));

switch (op)

{

case OpCode::ScopedEnsureNoRedeclFld:

case OpCode::ScopedDeleteFld:

case OpCode::ScopedDeleteFldStrict:

{

Output::Print(L" %s, R%d", pPropertyName->GetBuffer(), data->Value);

break;

}

case OpCode::ScopedInitFunc:

{

Output::Print(L" %s = R%d, R%d", pPropertyName->GetBuffer(), data->Value,

Js::FunctionBody::RootObjectRegSlot);

break;

}

default:

{

AssertMsg(false, "Unknown OpCode for OpLayoutElementScopedC");

break;

}

}

}

template <class T>

void ByteCodeDumper::DumpElementC(OpCode op, const unaligned T \* data, Js::FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

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

PropertyRecord const \* pPropertyName = scriptContext->GetPropertyName(

dumpFunction->GetReferencedPropertyId(data->PropertyIdIndex));

switch (op)

{

case OpCode::DeleteFld:

case OpCode::DeleteRootFld:

case OpCode::DeleteFldStrict:

case OpCode::DeleteRootFldStrict:

{

Output::Print(L" R%d.%s", data->Instance, pPropertyName->GetBuffer());

break;

}

case OpCode::InitSetFld:

case OpCode::InitGetFld:

case OpCode::InitClassMemberGet:

case OpCode::InitClassMemberSet:

{

Output::Print(L" R%d.%s = (Set/Get) R%d", data->Instance, pPropertyName->GetBuffer(),

data->Value);

break;

}

case OpCode::StFuncExpr:

case OpCode::InitProto:

{

Output::Print(L" R%d.%s = R%d", data->Instance, pPropertyName->GetBuffer(),

data->Value);

break;

}

default:

{

AssertMsg(false, "Unknown OpCode for OpLayoutElementC");

break;

}

}

}

template <class T>

void ByteCodeDumper::DumpElementScopedC2(OpCode op, const unaligned T \* data, Js::FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

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

PropertyRecord const \* pPropertyName = scriptContext->GetPropertyName(

dumpFunction->GetReferencedPropertyId(data->PropertyIdIndex));

switch (op)

{

case OpCode::ScopedLdInst:

{

Output::Print(L" R%d, R%d = %s", data->Value, data->Value2, pPropertyName->GetBuffer());

break;

}

default:

{

AssertMsg(false, "Unknown OpCode for OpLayoutElementScopedC2");

break;

}

}

}

template <class T>

void ByteCodeDumper::DumpElementC2(OpCode op, const unaligned T \* data, Js::FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

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

PropertyRecord const \* pPropertyName = scriptContext->GetPropertyName(

dumpFunction->GetReferencedPropertyId(data->PropertyIdIndex));

switch (op)

{

case OpCode::LdSuperFld:

{

Output::Print(L" R%d = R%d(this=R%d).%s #%d", data->Value, data->Instance, data->Value2,

pPropertyName->GetBuffer(), data->PropertyIdIndex);

break;

}

case OpCode::ProfiledLdSuperFld:

{

Output::Print(L" R%d = R%d(this=R%d).%s #%d", data->Value, data->Instance, data->Value2,

pPropertyName->GetBuffer(), data->PropertyIdIndex);

DumpProfileId(data->PropertyIdIndex);

break;

}

case OpCode::StSuperFld:

{

Output::Print(L" R%d.%s(this=R%d) = R%d #%d", data->Instance, pPropertyName->GetBuffer(),

data->Value2, data->Value, data->PropertyIdIndex);

break;

}

case OpCode::ProfiledStSuperFld:

{

Output::Print(L" R%d.%s(this=R%d) = R%d #%d", data->Instance, pPropertyName->GetBuffer(),

data->Value2, data->Value, data->PropertyIdIndex);

DumpProfileId(data->PropertyIdIndex);

break;

}

default:

{

AssertMsg(false, "Unknown OpCode for OpLayoutElementC2");

break;

}

}

}

template <class T>

void ByteCodeDumper::DumpReg1Unsigned1(OpCode op, const unaligned T\* data, Js::FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

switch (op)

{

case OpCode::InvalCachedScope:

#if ENABLE\_NATIVE\_CODEGEN

case OpCode::NewScopeSlots:

#endif

Output::Print(L" R%u[%u]", data->R0, data->C1);

break;

case OpCode::NewRegEx:

{

DumpReg(data->R0);

#if DBG

Output::Print(L"=");

UnifiedRegex::DebugWriter w;

dumpFunction->GetLiteralRegex(data->C1)->Print(&w);

#else

Output::Print(L"=<regex>");

#endif

break;

}

default:

DumpReg(data->R0);

Output::Print(L"=");

DumpU4(data->C1);

break;

};

}

template <class T>

void ByteCodeDumper::DumpElementSlot(OpCode op, const unaligned T \* data, Js::FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

switch (op)

{

case OpCode::NewInnerStackScFunc:

case OpCode::NewInnerScFunc:

case OpCode::NewInnerScGenFunc:

{

FunctionProxy\* pfuncActual = dumpFunction->GetNestedFunc((uint)data->SlotIndex)->GetFunctionProxy();

Output::Print(L" R%d = env:R%d, %s()", data->Value, data->Instance,

pfuncActual->EnsureDeserialized()->GetDisplayName());

break;

}

#if ENABLE\_NATIVE\_CODEGEN

case OpCode::StSlot:

case OpCode::StSlotChkUndecl:

#endif

case OpCode::StObjSlot:

case OpCode::StObjSlotChkUndecl:

Output::Print(L" R%d[%d] = R%d ",data->Instance,data->SlotIndex,data->Value);

break;

case OpCode::LdSlot:

#if ENABLE\_NATIVE\_CODEGEN

case OpCode::LdSlotArr:

#endif

case OpCode::LdObjSlot:

Output::Print(L" R%d = R%d[%d] ",data->Value,data->Instance,data->SlotIndex);

break;

default:

{

AssertMsg(false, "Unknown OpCode for OpLayoutElementSlot");

break;

}

}

}

template <class T>

void ByteCodeDumper::DumpElementSlotI1(OpCode op, const unaligned T \* data, Js::FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

switch (op)

{

case OpCode::StLocalSlot:

case OpCode::StLocalObjSlot:

case OpCode::StLocalSlotChkUndecl:

case OpCode::StLocalObjSlotChkUndecl:

Output::Print(L" [%d] = R%d ",data->SlotIndex, data->Value);

break;

case OpCode::LdLocalSlot:

case OpCode::LdEnvObj:

case OpCode::LdLocalObjSlot:

Output::Print(L" R%d = [%d] ",data->Value, data->SlotIndex);

break;

case OpCode::NewScFunc:

case OpCode::NewStackScFunc:

case OpCode::NewScGenFunc:

{

FunctionProxy\* pfuncActual = dumpFunction->GetNestedFunc((uint)data->SlotIndex)->GetFunctionProxy();

Output::Print(L" R%d = %s()", data->Value,

pfuncActual->EnsureDeserialized()->GetDisplayName());

break;

}

default:

{

AssertMsg(false, "Unknown OpCode for OpLayoutElementSlotI1");

break;

}

}

}

template <class T>

void ByteCodeDumper::DumpElementSlotI2(OpCode op, const unaligned T \* data, Js::FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

switch (op)

{

case OpCode::StInnerSlot:

case OpCode::StInnerObjSlot:

case OpCode::StEnvSlot:

case OpCode::StEnvObjSlot:

case OpCode::StEnvSlotChkUndecl:

case OpCode::StEnvObjSlotChkUndecl:

Output::Print(L" [%d][%d] = R%d ",data->SlotIndex1, data->SlotIndex2, data->Value);

break;

case OpCode::LdInnerSlot:

case OpCode::LdInnerObjSlot:

case OpCode::LdEnvSlot:

case OpCode::LdEnvObjSlot:

Output::Print(L" R%d = [%d][%d] ",data->Value, data->SlotIndex1, data->SlotIndex2);

break;

default:

{

AssertMsg(false, "Unknown OpCode for OpLayoutElementSlotI2");

break;

}

}

}

template <class T>

void ByteCodeDumper::DumpElementP(OpCode op, const unaligned T \* data, Js::FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

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

PropertyId propertyId = dumpFunction->GetPropertyIdFromCacheId(data->inlineCacheIndex);

PropertyRecord const \* pPropertyName = scriptContext->GetPropertyName(propertyId);

switch (op)

{

case OpCode::ScopedLdFldForTypeOf:

case OpCode::ScopedLdFld:

Output::Print(L" R%d = %s, R%d #%d", data->Value, pPropertyName->GetBuffer(),

Js::FunctionBody::RootObjectRegSlot, data->inlineCacheIndex);

break;

case OpCode::ScopedStFld:

case OpCode::ConsoleScopedStFld:

case OpCode::ScopedStFldStrict:

Output::Print(L" %s = R%d, R%d #%d", pPropertyName->GetBuffer(), data->Value,

Js::FunctionBody::RootObjectRegSlot, data->inlineCacheIndex);

break;

case OpCode::LdLocalFld:

Output::Print(L" R%d = %s #%d", data->Value, pPropertyName->GetBuffer(), data->inlineCacheIndex);

break;

case OpCode::ProfiledLdLocalFld:

Output::Print(L" R%d = %s #%d", data->Value, pPropertyName->GetBuffer(), data->inlineCacheIndex);

DumpProfileId(data->inlineCacheIndex);

break;

case OpCode::StLocalFld:

case OpCode::InitLocalFld:

case OpCode::InitLocalLetFld:

case OpCode::InitUndeclLocalLetFld:

case OpCode::InitUndeclLocalConstFld:

Output::Print(L" %s = R%d #%d", pPropertyName->GetBuffer(), data->Value, data->inlineCacheIndex);

break;

case OpCode::ProfiledStLocalFld:

case OpCode::ProfiledInitLocalFld:

Output::Print(L" %s = R%d #%d", pPropertyName->GetBuffer(), data->Value, data->inlineCacheIndex);

DumpProfileId(data->inlineCacheIndex);

break;

default:

{

AssertMsg(false, "Unknown OpCode for OpLayoutElementP");

break;

}

}

}

template <class T>

void ByteCodeDumper::DumpElementPIndexed(OpCode op, const unaligned T \* data, Js::FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

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

PropertyId propertyId = dumpFunction->GetPropertyIdFromCacheId(data->inlineCacheIndex);

PropertyRecord const \* pPropertyName = scriptContext->GetPropertyName(propertyId);

switch (op)

{

case OpCode::InitInnerFld:

case OpCode::InitInnerLetFld:

case OpCode::InitUndeclLetFld:

case OpCode::InitUndeclConstFld:

Output::Print(L" [%d].%s = R%d #%d", data->scopeIndex, pPropertyName->GetBuffer(), data->Value, data->inlineCacheIndex);

break;

default:

{

AssertMsg(false, "Unknown OpCode for OpLayoutElementPIndexed");

break;

}

}

}

template <class T>

void ByteCodeDumper::DumpElementCP(OpCode op, const unaligned T \* data, Js::FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

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

PropertyId propertyId = dumpFunction->GetPropertyIdFromCacheId(data->inlineCacheIndex);

PropertyRecord const \* pPropertyName = scriptContext->GetPropertyName(propertyId);

switch (op)

{

case OpCode::LdFldForTypeOf:

case OpCode::LdFld:

case OpCode::LdFldForCallApplyTarget:

case OpCode::LdMethodFld:

case OpCode::ScopedLdMethodFld:

{

Output::Print(L" R%d = R%d.%s #%d", data->Value, data->Instance,

pPropertyName->GetBuffer(), data->inlineCacheIndex);

break;

}

case OpCode::InitFld:

case OpCode::InitLetFld:

case OpCode::InitConstFld:

case OpCode::StFld:

case OpCode::StFldStrict:

case OpCode::InitClassMember:

{

Output::Print(L" R%d.%s = R%d #%d", data->Instance, pPropertyName->GetBuffer(),

data->Value, data->inlineCacheIndex);

break;

}

case OpCode::ProfiledLdFldForTypeOf:

case OpCode::ProfiledLdFld:

case OpCode::ProfiledLdFldForCallApplyTarget:

case OpCode::ProfiledLdMethodFld:

{

Output::Print(L" R%d = R%d.%s #%d", data->Value, data->Instance,

pPropertyName->GetBuffer(), data->inlineCacheIndex);

DumpProfileId(data->inlineCacheIndex);

break;

}

case OpCode::ProfiledInitFld:

case OpCode::ProfiledStFld:

case OpCode::ProfiledStFldStrict:

{

Output::Print(L" R%d.%s = R%d #%d", data->Instance, pPropertyName->GetBuffer(),

data->Value, data->inlineCacheIndex);

DumpProfileId(data->inlineCacheIndex);

break;

}

default:

{

AssertMsg(false, "Unknown OpCode for OpLayoutElementCP");

break;

}

}

}

template <class T>

void ByteCodeDumper::DumpElementRootCP(OpCode op, const unaligned T \* data, Js::FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

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

PropertyId propertyId = dumpFunction->GetPropertyIdFromCacheId(data->inlineCacheIndex);

PropertyRecord const \* pPropertyName = scriptContext->GetPropertyName(propertyId);

switch (op)

{

case OpCode::LdRootFld:

case OpCode::LdRootMethodFld:

case OpCode::LdRootFldForTypeOf:

{

Output::Print(L" R%d = root.%s #%d", data->Value,

pPropertyName->GetBuffer(), data->inlineCacheIndex);

break;

}

case OpCode::InitRootFld:

case OpCode::InitRootLetFld:

case OpCode::InitRootConstFld:

case OpCode::StRootFld:

case OpCode::StRootFldStrict:

{

Output::Print(L" root.%s = R%d #%d", pPropertyName->GetBuffer(),

data->Value, data->inlineCacheIndex);

break;

}

case OpCode::ProfiledLdRootFld:

case OpCode::ProfiledLdRootFldForTypeOf:

case OpCode::ProfiledLdRootMethodFld:

{

Output::Print(L" R%d = root.%s #%d", data->Value,

pPropertyName->GetBuffer(), data->inlineCacheIndex);

DumpProfileId(data->inlineCacheIndex);

break;

}

case OpCode::ProfiledInitRootFld:

case OpCode::ProfiledStRootFld:

case OpCode::ProfiledStRootFldStrict:

{

Output::Print(L" root.%s = R%d #%d", pPropertyName->GetBuffer(),

data->Value, data->inlineCacheIndex);

DumpProfileId(data->inlineCacheIndex);

break;

}

default:

{

AssertMsg(false, "Unknown OpCode for OpLayoutElementRootCP");

break;

}

}

}

template <class T>

void ByteCodeDumper::DumpElementUnsigned1(OpCode op, const unaligned T \* data, Js::FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

switch (op)

{

case OpCode::StArrItemC\_CI4:

case OpCode::StArrItemI\_CI4:

case OpCode::StArrSegItem\_CI4:

case OpCode::StArrInlineItem\_CI4:

Output::Print(L" R%d[", data->Instance);

DumpI4(data->Element);

Output::Print(L"] = R%d", data->Value);

break;

default:

AssertMsg(false, "Unknown OpCode for OpLayoutElementUnsigned1");

break;

}

}

template <class T>

void ByteCodeDumper::DumpArg(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

switch (op)

{

case OpCode::ProfiledArgOut\_A:

case OpCode::ArgOut\_A:

case OpCode::ArgOut\_ANonVar:

{

Output::Print(L" Out%d =", (int) data->Arg);

DumpReg(data->Reg);

break;

}

default:

{

AssertMsg(false, "Unknown OpCode for OpLayoutArg");

break;

}

}

}

template <class T>

void ByteCodeDumper::DumpArgNoSrc(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

switch (op)

{

case Js::OpCode::ArgOut\_Env:

{

Output::Print(L" Out%d ", (int) data->Arg);

break;

}

default:

{

AssertMsg(false, "Unknown OpCode for OpLayoutArgNoSrc");

break;

}

}

}

void

ByteCodeDumper::DumpStartCall(OpCode op, const unaligned OpLayoutStartCall \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

Assert(op == OpCode::StartCall );

Output::Print(L" ArgCount: %d", data->ArgCount);

}

template <class T> void

ByteCodeDumper::DumpUnsigned1(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpU4(data->C1);

}

template <class T> void

ByteCodeDumper::DumpReg1(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

switch (op)

{

case OpCode::ObjectFreeze:

Output::Print(L" R%d.freeze()", data->R0);

break;

default:

DumpReg(data->R0);

break;

}

}

template <class T> void

ByteCodeDumper::DumpReg2(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg(data->R0);

DumpReg(data->R1);

}

template <class T> void

ByteCodeDumper::DumpReg2WithICIndex(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg2(op, data, dumpFunction, reader);

Output::Print(L" <%d> ", data->inlineCacheIndex);

}

template <class T> void

ByteCodeDumper::DumpReg3(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

switch (op)

{

case OpCode::NewInnerScopeSlots:

Output::Print(L" [%d], %d, %d ", data->R0, data->R1, data->R2);

break;

default:

DumpReg(data->R0);

DumpReg(data->R1);

DumpReg(data->R2);

break;

}

}

template <class T> void

ByteCodeDumper::DumpReg3C(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

switch (op)

{

case OpCode::IsInst:

Output::Print(L"R%d = R%d instanceof R%d #%d",

data->R0, data->R1, data->R2, data->inlineCacheIndex);

break;

default:

AssertMsg(false, "Unknown OpCode for OpLayoutReg3C");

}

}

template <class T> void

ByteCodeDumper::DumpReg4(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg(data->R0);

DumpReg(data->R1);

DumpReg(data->R2);

DumpReg(data->R3);

}

template <class T> void

ByteCodeDumper::DumpReg2B1(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg(data->R0);

DumpReg(data->R1);

DumpI4(data->B2);

}

template <class T> void

ByteCodeDumper::DumpReg3B1(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg(data->R0);

DumpReg(data->R1);

DumpReg(data->R2);

DumpI4(data->B3);

}

template <class T> void

ByteCodeDumper::DumpReg5(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg(data->R0);

DumpReg(data->R1);

DumpReg(data->R2);

DumpReg(data->R3);

DumpReg(data->R4);

}

void

ByteCodeDumper::DumpW1(OpCode op, const unaligned OpLayoutW1 \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpU2(data->C1);

}

void

ByteCodeDumper::DumpReg1Int2(OpCode op, const unaligned OpLayoutReg1Int2 \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg(data->R0);

Output::Print(L"=");

DumpI4(data->C1);

DumpI4(data->C2);

}

void

ByteCodeDumper::DumpAuxNoReg(OpCode op, const unaligned OpLayoutAuxNoReg \* playout, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

switch (op)

{

case OpCode::CommitScope:

{

const Js::PropertyIdArray \*propIds = reader.ReadPropertyIdArray(playout->Offset, dumpFunction);

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

Output::Print(L" %d [", propIds->count);

for (uint i=0; i < propIds->count && i < 3; i++)

{

PropertyRecord const \* pPropertyName = scriptContext->GetPropertyName(propIds->elements[i]);

if (i != 0)

{

Output::Print(L", ");

}

Output::Print(L"%s", pPropertyName->GetBuffer());

}

Output::Print(L"]");

break;

}

case Js::OpCode::InitCachedFuncs:

{

const Js::FuncInfoArray \*arr = reader.ReadAuxArray<FuncInfoEntry>(playout->Offset, dumpFunction);

Output::Print(L" %d [", arr->count);

for (uint i = 0; i < arr->count && i < 3; i++)

{

Js::ParseableFunctionInfo \*info = dumpFunction->GetNestedFunctionForExecution(arr->elements[i].nestedIndex);

if (i != 0)

{

Output::Print(L", ");

}

Output::Print(L"%s", info->GetDisplayName());

}

Output::Print(L"]");

break;

}

default:

AssertMsg(false, "Unknown OpCode for OpLayoutType::AuxNoReg");

break;

}

}

void

ByteCodeDumper::DumpAuxiliary(OpCode op, const unaligned OpLayoutAuxiliary \* playout, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

switch (op)

{

case OpCode::NewScObjectLiteral:

case OpCode::LdPropIds:

{

const Js::PropertyIdArray \*propIds = reader.ReadPropertyIdArray(playout->Offset, dumpFunction);

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

DumpReg(playout->R0);

Output::Print(L"= %d [", propIds->count);

for (uint i=0; i< propIds->count && i < 3; i++)

{

PropertyRecord const \* pPropertyName = scriptContext->GetPropertyName(propIds->elements[i]);

if (i != 0)

{

Output::Print(L", ");

}

Output::Print(L"%s", pPropertyName->GetBuffer());

}

if (propIds->count >= 3)

{

Output::Print(L", ...");

}

Output::Print(L"], LiteralId %d", playout->C1);

break;

}

case OpCode::StArrSegItem\_A:

{

const Js::VarArray \*vars = reader.ReadAuxArray<Var>(playout->Offset, dumpFunction);

DumpReg(playout->R0);

Output::Print(L"= %d [", vars->count);

uint i=0;

for (; i<vars->count && i < 3; i++)

{

if (i != 0)

{

Output::Print(L", ");

}

Output::Print(L"%d", vars->elements[i]);

}

if (i != vars->count)

{

Output::Print(L", ...");

}

Output::Print(L"]");

break;

}

case OpCode::NewScIntArray:

{

const Js::AuxArray<int32> \*intArray = reader.ReadAuxArray<int32>(playout->Offset, dumpFunction);

Output::Print(L" R%d = %d [", playout->R0, intArray->count);

uint i;

for (i = 0; i<intArray->count && i < 3; i++)

{

if (i != 0)

{

Output::Print(L", ");

}

Output::Print(L"%d", intArray->elements[i]);

}

if (i != intArray->count)

{

Output::Print(L", ...");

}

Output::Print(L"]");

break;

}

case OpCode::NewScFltArray:

{

const Js::AuxArray<double> \*dblArray = reader.ReadAuxArray<double>(playout->Offset, dumpFunction);

Output::Print(L" R%d = %d [", playout->R0, dblArray->count);

uint i;

for (i = 0; i<dblArray->count && i < 3; i++)

{

if (i != 0)

{

Output::Print(L", ");

}

Output::Print(L"%f", dblArray->elements[i]);

}

if (i != dblArray->count)

{

Output::Print(L", ...");

}

Output::Print(L"]");

break;

}

case OpCode::NewScObject\_A:

{

const Js::VarArrayVarCount \*vars = reader.ReadVarArrayVarCount(playout->Offset, dumpFunction);

DumpReg(playout->R0);

int count = Js::TaggedInt::ToInt32(vars->count);

Output::Print(L"= %d [", count);

int i=0;

for (; i<count && i < 3; i++)

{

if (i != 0)

{

Output::Print(L", ");

}

if (TaggedInt::Is(vars->elements[i]))

{

Output::Print(L"%d", TaggedInt::ToInt32(vars->elements[i]));

}

else if (JavascriptNumber::Is(vars->elements[i]))

{

Output::Print(L"%g", JavascriptNumber::GetValue(vars->elements[i]));

}

else

{

Assert(false);

}

}

if (i != count)

{

Output::Print(L", ...");

}

Output::Print(L"]");

break;

}

default:

AssertMsg(false, "Unknown OpCode for OpLayoutType::Auxiliary");

break;

}

}

void

ByteCodeDumper::DumpReg2Aux(OpCode op, const unaligned OpLayoutReg2Aux \* playout, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

switch (op)

{

case Js::OpCode::SpreadArrayLiteral:

{

const Js::AuxArray<uint32> \*arr = reader.ReadAuxArray<uint32>(playout->Offset, dumpFunction);

Output::Print(L" R%u <- R%u, %u spreadArgs [", playout->R0, playout->R1, arr->count);

for (uint i = 0; i < arr->count; i++)

{

if (i > 10)

{

Output::Print(L", ...");

break;

}

if (i != 0)

{

Output::Print(L", ");

}

Output::Print(L"%u", arr->elements[i]);

}

Output::Print(L"]");

break;

}

default:

AssertMsg(false, "Unknown OpCode for OpLayoutType::Reg2Aux");

break;

}

}

template <class T>

void ByteCodeDumper::DumpClass(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpReg(data->Constructor);

if (data->Extends != Js::Constants::NoRegister)

{

Output::Print(L"extends");

DumpReg((RegSlot)data->Extends);

}

}

#ifdef BYTECODE\_BRANCH\_ISLAND

void ByteCodeDumper::DumpBrLong(OpCode op, const unaligned OpLayoutBrLong\* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpOffset(data->RelativeJumpOffset, reader);

}

#endif

void ByteCodeDumper::DumpBr(OpCode op, const unaligned OpLayoutBr \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpOffset(data->RelativeJumpOffset, reader);

}

void ByteCodeDumper::DumpBrS(OpCode op, const unaligned OpLayoutBrS \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpOffset(data->RelativeJumpOffset, reader);

DumpI4(data->val);

}

template <class T>

void ByteCodeDumper::DumpBrReg1(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpOffset(data->RelativeJumpOffset, reader);

DumpReg(data->R1);

}

template <class T>

void ByteCodeDumper::DumpBrReg2(OpCode op, const unaligned T \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpOffset(data->RelativeJumpOffset, reader);

DumpReg(data->R1);

DumpReg(data->R2);

}

void ByteCodeDumper::DumpBrProperty(OpCode op, const unaligned OpLayoutBrProperty \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpOffset(data->RelativeJumpOffset, reader);

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

PropertyRecord const \* pPropertyName = scriptContext->GetPropertyName(

dumpFunction->GetReferencedPropertyId(data->PropertyIdIndex));

Output::Print(L"R%d.%s", data->Instance, pPropertyName->GetBuffer());

}

void ByteCodeDumper::DumpBrLocalProperty(OpCode op, const unaligned OpLayoutBrLocalProperty \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpOffset(data->RelativeJumpOffset, reader);

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

PropertyRecord const \* pPropertyName = scriptContext->GetPropertyName(

dumpFunction->GetReferencedPropertyId(data->PropertyIdIndex));

Output::Print(L"%s", pPropertyName->GetBuffer());

}

void ByteCodeDumper::DumpBrEnvProperty(OpCode op, const unaligned OpLayoutBrEnvProperty \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader)

{

DumpOffset(data->RelativeJumpOffset, reader);

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

PropertyRecord const \* pPropertyName = scriptContext->GetPropertyName(

dumpFunction->GetReferencedPropertyId(data->PropertyIdIndex));

Output::Print(L"[%d].%s", data->SlotIndex, pPropertyName->GetBuffer());

}

} // namespace Js

#endif

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

// 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

#if DBG\_DUMP

namespace Js

{

struct ByteCodeDumper /\* All static \*/

{

public:

static void DumpRecursively(FunctionBody\* dumpFunction);

static void Dump(FunctionBody \* dumpFunction);

static void DumpConstantTable(FunctionBody \*dumpFunction);

static void DumpOp(OpCode op, LayoutSize layoutSize, ByteCodeReader& reader, FunctionBody \* dumpFunction);

protected:

static void DumpImplicitArgIns(FunctionBody \* dumpFunction);

static void DumpI4(int value);

static void DumpU4(uint32 value);

static void DumpU2(uint16 value);

static void DumpOffset(int byteOffset, ByteCodeReader const& reader);

static void DumpAddr(void\* addr);

static void DumpR8(double value);

static void DumpReg(RegSlot registerID);

static void DumpReg(RegSlot\_TwoByte registerID);

static void DumpReg(RegSlot\_OneByte registerID);

static void DumpProfileId(uint id);

#define LAYOUT\_TYPE(layout) \

static void Dump##layout(OpCode op, const unaligned OpLayout##layout\* data, FunctionBody \* dumpFunction, ByteCodeReader& reader);

#define LAYOUT\_TYPE\_WMS(layout) \

template <class T> static void Dump##layout(OpCode op, const unaligned T\* data, FunctionBody \* dumpFunction, ByteCodeReader& reader);

#define LAYOUT\_TYPE\_PROFILED(layout) \

LAYOUT\_TYPE(layout) \

static void DumpProfiled##layout(OpCode op, const unaligned OpLayoutProfiled##layout \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader) \

{ \

Assert(OpCodeUtil::GetOpCodeLayout(op) == OpLayoutType::Profiled##layout); \

Js::OpCodeUtil::ConvertOpToNonProfiled(op); \

Dump##layout(op, data, dumpFunction, reader); \

DumpProfileId(data->profileId); \

}

#define LAYOUT\_TYPE\_PROFILED2(layout) \

LAYOUT\_TYPE(layout) \

static void DumpProfiled2##layout(OpCode op, const unaligned OpLayoutProfiled2##layout \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader) \

{ \

Assert(OpCodeUtil::GetOpCodeLayout(op) == OpLayoutType::Profiled2##layout); \

Js::OpCodeUtil::ConvertOpToNonProfiled(op); \

Dump##layout(op, data, dumpFunction, reader); \

DumpProfileId(data->profileId); \

DumpProfileId(data->profileId2); \

}

#define LAYOUT\_TYPE\_PROFILED\_WMS(layout) \

LAYOUT\_TYPE\_WMS(layout) \

template <class T> \

static void DumpProfiled##layout(OpCode op, const unaligned OpLayoutDynamicProfile<T> \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader) \

{ \

Assert(OpCodeUtil::GetOpCodeLayout(op) == OpLayoutType::Profiled##layout); \

Js::OpCodeUtil::ConvertOpToNonProfiled(op); \

Dump##layout<T>(op, data, dumpFunction, reader); \

DumpProfileId(data->profileId); \

}

#define LAYOUT\_TYPE\_PROFILED2\_WMS(layout) \

LAYOUT\_TYPE\_PROFILED\_WMS(layout) \

template <class T> \

static void DumpProfiled2##layout(OpCode op, const unaligned OpLayoutDynamicProfile2<T> \* data, FunctionBody \* dumpFunction, ByteCodeReader& reader) \

{ \

Assert(OpCodeUtil::GetOpCodeLayout(op) == OpLayoutType::Profiled2##layout); \

Js::OpCodeUtil::ConvertOpToNonProfiled(op); \

Dump##layout<T>(op, data, dumpFunction, reader); \

DumpProfileId(data->profileId); \

DumpProfileId(data->profileId2); \

}

#include "LayoutTypes.h"

};

} // namespace Js

#endif

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

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

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

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

#include "RuntimeByteCodePch.h"

#include "FormalsUtil.h"

#include "Language\AsmJs.h"

void EmitReference(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo);

void EmitAssignment(ParseNode \*asgnNode, ParseNode \*lhs, Js::RegSlot rhsLocation, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo);

void EmitLoad(ParseNode \*rhs, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo);

void EmitCall(ParseNode\* pnode, Js::RegSlot rhsLocation, ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* funcInfo, BOOL fReturnValue, BOOL fEvaluateComponents, BOOL fHasNewTarget, Js::RegSlot overrideThisLocation = Js::Constants::NoRegister);

void EmitSuperFieldPatch(FuncInfo\* funcInfo, ParseNode\* pnode, ByteCodeGenerator\* byteCodeGenerator);

bool EmitUseBeforeDeclaration(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo);

void EmitUseBeforeDeclarationRuntimeError(ByteCodeGenerator \*byteCodeGenerator, Js::RegSlot location, bool fLoadUndef = true);

void VisitClearTmpRegs(ParseNode \* pnode, ByteCodeGenerator \* byteCodeGenerator, FuncInfo \* funcInfo);

bool CallTargetIsArray(ParseNode \*pnode)

{

return pnode->nop == knopName && pnode->sxPid.PropertyIdFromNameNode() == Js::PropertyIds::Array;

}

BOOL MayHaveSideEffectOnNode(ParseNode \*pnode, ParseNode \*pnodeSE)

{

// Try to determine whether pnodeSE may kill the named var represented by pnode.

if (pnode->nop == knopComputedName)

{

pnode = pnode->sxUni.pnode1;

}

if (pnode->nop != knopName)

{

// Only investigating named vars here.

return false;

}

uint fnop = ParseNode::Grfnop(pnodeSE->nop);

if (fnop & fnopLeaf)

{

// pnodeSE is a leaf and can't kill anything.

return false;

}

if (fnop & fnopAsg)

{

// pnodeSE is an assignment (=, ++, +=, etc.)

// Trying to examine the LHS of pnodeSE caused small perf regressions,

// maybe because of code layout or some other subtle effect.

return true;

}

if (fnop & fnopUni)

{

// pnodeSE is a unary op, so recurse to the source (if present - e.g., [] may have no opnd).

if (pnodeSE->nop == knopTempRef)

{

return false;

}

else

{

return pnodeSE->sxUni.pnode1 && MayHaveSideEffectOnNode(pnode, pnodeSE->sxUni.pnode1);

}

}

else if (fnop & fnopBin)

{

// pnodeSE is a binary (or ternary) op, so recurse to the sources (if present).

if (pnodeSE->nop == knopQmark)

{

return MayHaveSideEffectOnNode(pnode, pnodeSE->sxTri.pnode1) ||

MayHaveSideEffectOnNode(pnode, pnodeSE->sxTri.pnode2) ||

MayHaveSideEffectOnNode(pnode, pnodeSE->sxTri.pnode3);

}

else if (pnodeSE->nop == knopCall || pnodeSE->nop == knopNew)

{

return MayHaveSideEffectOnNode(pnode, pnodeSE->sxCall.pnodeTarget) ||

(pnodeSE->sxCall.pnodeArgs && MayHaveSideEffectOnNode(pnode, pnodeSE->sxCall.pnodeArgs));

}

else

{

return MayHaveSideEffectOnNode(pnode, pnodeSE->sxBin.pnode1) ||

(pnodeSE->sxBin.pnode2 && MayHaveSideEffectOnNode(pnode, pnodeSE->sxBin.pnode2));

}

}

else if (pnodeSE->nop == knopList)

{

return true;

}

return false;

}

bool IsCallOfConstants(ParseNode \*pnode);

bool BlockHasOwnScope(ParseNode\* pnodeBlock, ByteCodeGenerator \*byteCodeGenerator);

bool CreateNativeArrays(ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo);

bool IsArguments(ParseNode \*pnode)

{

for (;;)

{

switch (pnode->nop)

{

case knopName:

return pnode->sxPid.sym && pnode->sxPid.sym->GetIsArguments();

case knopCall:

case knopNew:

if (IsArguments(pnode->sxCall.pnodeTarget))

{

return true;

}

if (pnode->sxCall.pnodeArgs)

{

ParseNode \*pnodeArg = pnode->sxCall.pnodeArgs;

while (pnodeArg->nop == knopList)

{

if (IsArguments(pnodeArg->sxBin.pnode1))

return true;

pnodeArg = pnodeArg->sxBin.pnode2;

}

pnode = pnodeArg;

break;

}

return false;

case knopArray:

if (pnode->sxArrLit.arrayOfNumbers || pnode->sxArrLit.count == 0)

{

return false;

}

pnode = pnode->sxUni.pnode1;

break;

case knopQmark:

if (IsArguments(pnode->sxTri.pnode1) || IsArguments(pnode->sxTri.pnode2))

{

return true;

}

pnode = pnode->sxTri.pnode3;

break;

//

// Cases where we don't check for "arguments" yet.

// Assume that they might have it. Disable the optimization is such scenarios

//

case knopList:

case knopObject:

case knopVarDecl:

case knopConstDecl:

case knopLetDecl:

case knopFncDecl:

case knopClassDecl:

case knopFor:

case knopIf:

case knopDoWhile:

case knopWhile:

case knopForIn:

case knopForOf:

case knopReturn:

case knopBlock:

case knopBreak:

case knopContinue:

case knopLabel:

case knopTypeof:

case knopThrow:

case knopWith:

case knopFinally:

case knopTry:

case knopTryCatch:

case knopTryFinally:

return true;

default:

{

uint flags = ParseNode::Grfnop(pnode->nop);

if (flags&fnopUni)

{

Assert(pnode->sxUni.pnode1);

pnode = pnode->sxUni.pnode1;

break;

}

else if (flags&fnopBin)

{

Assert(pnode->sxBin.pnode1 && pnode->sxBin.pnode2);

if (IsArguments(pnode->sxBin.pnode1))

{

return true;

}

pnode = pnode->sxBin.pnode2;

break;

}

return false;

}

}

}

}

bool ApplyEnclosesArgs(ParseNode\* fncDecl, ByteCodeGenerator\* byteCodeGenerator);

void Emit(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo, BOOL fReturnValue, bool isConstructorCall = false, ParseNode \*bindPnode = nullptr);

void EmitComputedFunctionNameVar(ParseNode \*nameNode, ParseNode \*exprNode, ByteCodeGenerator \*byteCodeGenerator);

void EmitBinaryOpnds(ParseNode \*pnode1, ParseNode \*pnode2, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo);

bool IsExpressionStatement(ParseNode\* stmt, const Js::ScriptContext \*const scriptContext);

void EmitInvoke(Js::RegSlot location, Js::RegSlot callObjLocation, Js::PropertyId propertyId, ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* funcInfo);

void EmitInvoke(Js::RegSlot location, Js::RegSlot callObjLocation, Js::PropertyId propertyId, ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* funcInfo, Js::RegSlot arg1Location);

static const Js::OpCode nopToOp[knopLim] =

{

#define OP(x) Br##x##\_A

#define PTNODE(nop,sn,pc,nk,grfnop,json) Js::OpCode::pc,

#include "ptlist.h"

};

static const Js::OpCode nopToCMOp[knopLim] =

{

#define OP(x) Cm##x##\_A

#define PTNODE(nop,sn,pc,nk,grfnop,json) Js::OpCode::pc,

#include "ptlist.h"

};

Js::OpCode ByteCodeGenerator::ToChkUndeclOp(Js::OpCode op) const

{

switch(op)

{

case Js::OpCode::StLocalSlot:

return Js::OpCode::StLocalSlotChkUndecl;

case Js::OpCode::StInnerSlot:

return Js::OpCode::StInnerSlotChkUndecl;

case Js::OpCode::StEnvSlot:

return Js::OpCode::StEnvSlotChkUndecl;

case Js::OpCode::StObjSlot:

return Js::OpCode::StObjSlotChkUndecl;

case Js::OpCode::StLocalObjSlot:

return Js::OpCode::StLocalObjSlotChkUndecl;

case Js::OpCode::StEnvObjSlot:

return Js::OpCode::StEnvObjSlotChkUndecl;

default:

AssertMsg(false, "Unknown opcode for chk undecl mapping");

return Js::OpCode::InvalidOpCode;

}

}

// Tracks a register slot let/const property for the passed in debugger block/catch scope.

// debuggerScope - The scope to add the variable to.

// symbol - The symbol that represents the register property.

// funcInfo - The function info used to store the property into the tracked debugger register slot list.

// flags - The flags to assign to the property.

// isFunctionDeclaration - Whether or not the register is a function declaration, which requires that its byte code offset be updated immediately.

void ByteCodeGenerator::TrackRegisterPropertyForDebugger(

Js::DebuggerScope \*debuggerScope,

Symbol \*symbol,

FuncInfo \*funcInfo,

Js::DebuggerScopePropertyFlags flags /\*= Js::DebuggerScopePropertyFlags\_None\*/,

bool isFunctionDeclaration /\*= false\*/)

{

Assert(debuggerScope);

Assert(symbol);

Assert(funcInfo);

Js::RegSlot location = symbol->GetLocation();

Js::DebuggerScope \*correctDebuggerScope = debuggerScope;

if (debuggerScope->scopeType != Js::DiagExtraScopesType::DiagBlockScopeDirect && debuggerScope->scopeType != Js::DiagExtraScopesType::DiagCatchScopeDirect)

{

// We have to get the appropriate scope and add property over there.

// Make sure the scope is created whether we're in debug mode or not, because we

// need the empty scopes present during reparsing for debug mode.

correctDebuggerScope = debuggerScope->GetSiblingScope(location, Writer()->GetFunctionWrite());

}

if (this->ShouldTrackDebuggerMetadata() && !symbol->GetIsTrackedForDebugger())

{

// Only track the property if we're in debug mode since it's only needed by the debugger.

Js::PropertyId propertyId = symbol->EnsurePosition(this);

this->Writer()->AddPropertyToDebuggerScope(

correctDebuggerScope,

location,

propertyId,

/\*shouldConsumeRegister\*/ true,

flags,

isFunctionDeclaration);

Js::FunctionBody \*byteCodeFunction = funcInfo->GetParsedFunctionBody();

byteCodeFunction->InsertSymbolToRegSlotList(location, propertyId, funcInfo->varRegsCount);

symbol->SetIsTrackedForDebugger(true);

}

}

void ByteCodeGenerator::TrackActivationObjectPropertyForDebugger(

Js::DebuggerScope \*debuggerScope,

Symbol \*symbol,

Js::DebuggerScopePropertyFlags flags /\*= Js::DebuggerScopePropertyFlags\_None\*/,

bool isFunctionDeclaration /\*= false\*/)

{

Assert(debuggerScope);

Assert(symbol);

// Only need to track activation object properties in debug mode.

if (ShouldTrackDebuggerMetadata() && !symbol->GetIsTrackedForDebugger())

{

Js::RegSlot location = symbol->GetLocation();

Js::PropertyId propertyId = symbol->EnsurePosition(this);

this->Writer()->AddPropertyToDebuggerScope(

debuggerScope,

location,

propertyId,

/\*shouldConsumeRegister\*/ false,

flags,

isFunctionDeclaration);

symbol->SetIsTrackedForDebugger(true);

}

}

void ByteCodeGenerator::TrackSlotArrayPropertyForDebugger(

Js::DebuggerScope \*debuggerScope,

Symbol\* symbol,

Js::PropertyId propertyId,

Js::DebuggerScopePropertyFlags flags /\*= Js::DebuggerScopePropertyFlags\_None\*/,

bool isFunctionDeclaration /\*= false\*/)

{

// Note: Slot array properties are tracked even in non-debug mode in order to support slot array serialization

// of let/const variables between non-debug and debug mode (for example, when a slot array var escapes and is retrieved

// after a debugger attach or for WWA apps). They are also needed for heap enumeration.

Assert(debuggerScope);

Assert(symbol);

if (!symbol->GetIsTrackedForDebugger())

{

Js::RegSlot location = symbol->GetScopeSlot();

Assert(location != Js::Constants::NoRegister);

Assert(propertyId != Js::Constants::NoProperty);

this->Writer()->AddPropertyToDebuggerScope(

debuggerScope,

location,

propertyId,

/\*shouldConsumeRegister\*/ false,

flags,

isFunctionDeclaration);

symbol->SetIsTrackedForDebugger(true);

}

}

// Tracks a function declaration inside a block scope for the debugger metadata's current scope (let binding).

void ByteCodeGenerator::TrackFunctionDeclarationPropertyForDebugger(Symbol \*functionDeclarationSymbol, FuncInfo \*funcInfoParent)

{

Assert(functionDeclarationSymbol);

Assert(funcInfoParent);

AssertMsg(functionDeclarationSymbol->GetIsBlockVar(), "We should only track inner function let bindings for the debugger.");

// Note: we don't have to check symbol->GetIsTrackedForDebugger, as we are not doing actual work here,

// which is done in other Track\* functions that we call.

if (functionDeclarationSymbol->IsInSlot(funcInfoParent))

{

if (functionDeclarationSymbol->GetScope()->GetIsObject())

{

this->TrackActivationObjectPropertyForDebugger(

this->Writer()->GetCurrentDebuggerScope(),

functionDeclarationSymbol,

Js::DebuggerScopePropertyFlags\_None,

true /\*isFunctionDeclaration\*/);

}

else

{

// Make sure the property has a slot. This will bump up the size of the slot array if necessary.

// Note that slot array inner function bindings are tracked even in non-debug mode in order

// to keep the lifetime of the closure binding that could escape around for heap enumeration.

functionDeclarationSymbol->EnsureScopeSlot(funcInfoParent);

functionDeclarationSymbol->EnsurePosition(this);

this->TrackSlotArrayPropertyForDebugger(

this->Writer()->GetCurrentDebuggerScope(),

functionDeclarationSymbol,

functionDeclarationSymbol->GetPosition(),

Js::DebuggerScopePropertyFlags\_None,

true /\*isFunctionDeclaration\*/);

}

}

else

{

this->TrackRegisterPropertyForDebugger(

this->Writer()->GetCurrentDebuggerScope(),

functionDeclarationSymbol,

funcInfoParent,

Js::DebuggerScopePropertyFlags\_None,

true /\*isFunctionDeclaration\*/);

}

}

// Updates the byte code offset of the property with the passed in location and ID.

// Used to track let/const variables that are in the dead zone debugger side.

// location - The activation object, scope slot index, or register location for the property.

// propertyId - The ID of the property to update.

// shouldConsumeRegister - Whether or not the a register should be consumed (used for reg slot locations).

void ByteCodeGenerator::UpdateDebuggerPropertyInitializationOffset(Js::RegSlot location, Js::PropertyId propertyId, bool shouldConsumeRegister)

{

Assert(this->Writer());

Js::DebuggerScope\* currentDebuggerScope = this->Writer()->GetCurrentDebuggerScope();

Assert(currentDebuggerScope);

if (currentDebuggerScope != nullptr)

{

this->Writer()->UpdateDebuggerPropertyInitializationOffset(

currentDebuggerScope,

location,

propertyId,

shouldConsumeRegister);

}

}

void ByteCodeGenerator::LoadHeapArguments(FuncInfo \*funcInfo)

{

if (funcInfo->GetHasCachedScope())

{

this->LoadCachedHeapArguments(funcInfo);

}

else

{

this->LoadUncachedHeapArguments(funcInfo);

}

}

void GetFormalArgsArray(ByteCodeGenerator \*byteCodeGenerator, FuncInfo \* funcInfo, Js::PropertyIdArray \*propIds)

{

Assert(funcInfo);

Assert(propIds);

Assert(byteCodeGenerator);

bool hadDuplicates = false;

Js::ArgSlot i = 0;

auto processArg = [&](ParseNode \*pnode)

{

if (pnode->IsVarLetOrConst())

{

Assert(i < propIds->count);

Symbol \*sym = pnode->sxVar.sym;

Assert(sym);

Js::PropertyId symPos = sym->EnsurePosition(byteCodeGenerator);

//

// Check if the function has any same name parameters

// For the same name param, only the last one will be passed the correct propertyid

// For remaining dup param names, pass Constants::NoProperty

//

for (Js::ArgSlot j = 0; j < i; j++)

{

if (propIds->elements[j] == symPos)

{

// Found a dup parameter name

propIds->elements[j] = Js::Constants::NoProperty;

hadDuplicates = true;

break;

}

}

propIds->elements[i] = symPos;

}

else

{

propIds->elements[i] = Js::Constants::NoProperty;

}

++i;

};

MapFormals(funcInfo->root, processArg);

propIds->hadDuplicates = hadDuplicates;

}

void ByteCodeGenerator::LoadUncachedHeapArguments(FuncInfo \*funcInfo)

{

Assert(funcInfo->GetHasHeapArguments());

Scope \*scope = funcInfo->GetBodyScope();

Assert(scope);

Symbol \*argSym = funcInfo->GetArgumentsSymbol();

Assert(argSym && argSym->GetIsArguments());

Js::RegSlot argumentsLoc = argSym->GetLocation();

Js::RegSlot propIdsLoc = funcInfo->nullConstantRegister;

Js::OpCode opcode = funcInfo->root->sxFnc.IsSimpleParameterList() ? Js::OpCode::LdHeapArguments : Js::OpCode::LdLetHeapArguments;

bool hasRest = funcInfo->root->sxFnc.pnodeRest != nullptr;

uint count = funcInfo->inArgsCount + (hasRest ? 1 : 0) - 1;

if (count == 0)

{

// If no formals to function (only "this"), then no need to create the scope object.

// Leave both the arguments location and the propertyIds location as null.

Assert(funcInfo->root->sxFnc.pnodeArgs == nullptr && !hasRest);

}

else if (!NeedScopeObjectForArguments(funcInfo, funcInfo->root))

{

// We may not need a scope object for arguments, e.g. strict mode with no eval.

}

else if (funcInfo->frameObjRegister != Js::Constants::NoRegister)

{

// Pass the frame object and ID array to the runtime, and put the resulting Arguments object

// at the expected location.

propIdsLoc = argumentsLoc;

Js::PropertyIdArray \*propIds = AnewPlus(GetAllocator(), count \* sizeof(Js::PropertyId), Js::PropertyIdArray, count);

GetFormalArgsArray(this, funcInfo, propIds);

// Generate the opcode with propIds

Writer()->Auxiliary(

Js::OpCode::LdPropIds,

propIdsLoc,

propIds,

sizeof(Js::PropertyIdArray) + count \* sizeof(Js::PropertyId),

count);

AdeletePlus(GetAllocator(), count \* sizeof(Js::PropertyId), propIds);

}

this->m\_writer.Reg2(opcode, argumentsLoc, propIdsLoc);

EmitLocalPropInit(argSym->GetLocation(), argSym, funcInfo);

}

void ByteCodeGenerator::LoadCachedHeapArguments(FuncInfo \*funcInfo)

{

Assert(funcInfo->GetHasHeapArguments());

Scope \*scope = funcInfo->GetBodyScope();

Assert(scope);

Symbol \*argSym = funcInfo->GetArgumentsSymbol();

Assert(argSym && argSym->GetIsArguments());

Js::RegSlot argumentsLoc = argSym->GetLocation();

Js::OpCode op = funcInfo->root->sxFnc.IsSimpleParameterList() ? Js::OpCode::LdHeapArgsCached : Js::OpCode::LdLetHeapArgsCached;

this->m\_writer.Reg1(op, argumentsLoc);

EmitLocalPropInit(argumentsLoc, argSym, funcInfo);

}

Js::JavascriptArray\* ByteCodeGenerator::BuildArrayFromStringList(ParseNode\* stringNodeList, uint arrayLength, Js::ScriptContext\* scriptContext)

{

Assert(stringNodeList);

uint index = 0;

Js::Var str;

IdentPtr pid;

Js::JavascriptArray\* pArr = scriptContext->GetLibrary()->CreateArray(arrayLength);

while (stringNodeList->nop == knopList)

{

Assert(stringNodeList->sxBin.pnode1->nop == knopStr);

pid = stringNodeList->sxBin.pnode1->sxPid.pid;

str = Js::JavascriptString::NewCopyBuffer(pid->Psz(), pid->Cch(), scriptContext);

pArr->SetItemWithAttributes(index, str, PropertyEnumerable);

stringNodeList = stringNodeList->sxBin.pnode2;

index++;

}

Assert(stringNodeList->nop == knopStr);

pid = stringNodeList->sxPid.pid;

str = Js::JavascriptString::NewCopyBuffer(pid->Psz(), pid->Cch(), scriptContext);

pArr->SetItemWithAttributes(index, str, PropertyEnumerable);

return pArr;

}

// For now, this just assigns field ids for the current script.

// Later, we will combine this information with the global field id map.

// This temporary code will not work if a global member is accessed both with and without a LHS.

void ByteCodeGenerator::AssignPropertyIds(Js::ParseableFunctionInfo\* functionInfo)

{

globalScope->ForEachSymbol([this, functionInfo](Symbol \* sym)

{

this->AssignPropertyId(sym, functionInfo);

});

}

void ByteCodeGenerator::InitBlockScopedContent(ParseNode \*pnodeBlock, Js::DebuggerScope\* debuggerScope, FuncInfo \*funcInfo)

{

Assert(pnodeBlock->nop == knopBlock);

auto genBlockInit = [this, debuggerScope, funcInfo](ParseNode \*pnode)

{

// Only check if the scope is valid when let/const vars are in the scope. If there are no let/const vars,

// the debugger scope will not be created.

AssertMsg(debuggerScope, "Missing a case of scope tracking in BeginEmitBlock.");

FuncInfo \*funcInfo = this->TopFuncInfo();

Symbol \*sym = pnode->sxVar.sym;

Scope \*scope = sym->GetScope();

if (sym->GetIsGlobal())

{

Js::PropertyId propertyId = sym->EnsurePosition(this);

if (this->flags & fscrEval)

{

AssertMsg(this->IsConsoleScopeEval(), "Let/Consts cannot be in global scope outside of console eval");

Js::OpCode op = (sym->GetDecl()->nop == knopConstDecl) ? Js::OpCode::InitUndeclConsoleConstFld : Js::OpCode::InitUndeclConsoleLetFld;

this->m\_writer.ElementScopedU(op, funcInfo->FindOrAddReferencedPropertyId(propertyId));

}

else

{

Js::OpCode op = (sym->GetDecl()->nop == knopConstDecl) ?

Js::OpCode::InitUndeclRootConstFld : Js::OpCode::InitUndeclRootLetFld;

this->m\_writer.ElementRootU(op, funcInfo->FindOrAddReferencedPropertyId(propertyId));

}

}

else if (sym->IsInSlot(funcInfo))

{

if (scope->GetIsObject())

{

Js::RegSlot scopeLocation = scope->GetLocation();

Js::PropertyId propertyId = sym->EnsurePosition(this);

if (scopeLocation != Js::Constants::NoRegister && scopeLocation == funcInfo->frameObjRegister)

{

uint cacheId = funcInfo->FindOrAddInlineCacheId(scopeLocation, propertyId, false, true);

Js::OpCode op = (sym->GetDecl()->nop == knopConstDecl) ?

Js::OpCode::InitUndeclLocalConstFld : Js::OpCode::InitUndeclLocalLetFld;

this->m\_writer.ElementP(op, ByteCodeGenerator::ReturnRegister, cacheId);

}

else

{

uint cacheId = funcInfo->FindOrAddInlineCacheId(funcInfo->InnerScopeToRegSlot(scope), propertyId, false, true);

Js::OpCode op = (sym->GetDecl()->nop == knopConstDecl) ?

Js::OpCode::InitUndeclConstFld : Js::OpCode::InitUndeclLetFld;

this->m\_writer.ElementPIndexed(op, ByteCodeGenerator::ReturnRegister, scope->GetInnerScopeIndex(), cacheId);

}

TrackActivationObjectPropertyForDebugger(debuggerScope, sym, pnode->nop == knopConstDecl ? Js::DebuggerScopePropertyFlags\_Const : Js::DebuggerScopePropertyFlags\_None);

}

else if (CONFIG\_FLAG(TDZ))

{

Js::RegSlot tmpReg = funcInfo->AcquireTmpRegister();

this->m\_writer.Reg1(Js::OpCode::InitUndecl, tmpReg);

this->EmitLocalPropInit(tmpReg, sym, funcInfo);

funcInfo->ReleaseTmpRegister(tmpReg);

// Slot array properties are tracked in non-debug mode as well because they need to stay

// around for heap enumeration and escaping during attach/detach.

TrackSlotArrayPropertyForDebugger(debuggerScope, sym, sym->EnsurePosition(this), pnode->nop == knopConstDecl ? Js::DebuggerScopePropertyFlags\_Const : Js::DebuggerScopePropertyFlags\_None);

}

}

else

{

if (sym->GetDecl()->sxVar.isSwitchStmtDecl && CONFIG\_FLAG(TDZ))

{

// let/const declared in a switch is the only case of a variable that must be checked for

// use-before-declaration dynamically within its own function.

this->m\_writer.Reg1(Js::OpCode::InitUndecl, sym->GetLocation());

}

// Syms that begin in register may be delay-captured. In debugger mode, such syms

// will live only in slots, so tell the debugger to find them there.

if (sym->NeedsSlotAlloc(funcInfo))

{

TrackSlotArrayPropertyForDebugger(debuggerScope, sym, sym->EnsurePosition(this), pnode->nop == knopConstDecl ? Js::DebuggerScopePropertyFlags\_Const : Js::DebuggerScopePropertyFlags\_None);

}

else

{

TrackRegisterPropertyForDebugger(debuggerScope, sym, funcInfo, pnode->nop == knopConstDecl ? Js::DebuggerScopePropertyFlags\_Const : Js::DebuggerScopePropertyFlags\_None);

}

}

};

IterateBlockScopedVariables(pnodeBlock, genBlockInit);

}

// Records the start of a debugger scope if the passed in node has any let/const variables (or is not a block node).

// If it has no let/const variables, nullptr will be returned as no scope will be created.

Js::DebuggerScope\* ByteCodeGenerator::RecordStartScopeObject(ParseNode \*pnodeBlock, Js::DiagExtraScopesType scopeType, Js::RegSlot scopeLocation /\*= Js::Constants::NoRegister\*/, int\* index /\*= nullptr\*/)

{

Assert(pnodeBlock);

if (pnodeBlock->nop == knopBlock && !pnodeBlock->sxBlock.HasBlockScopedContent())

{

// In order to reduce allocations now that we track debugger scopes in non-debug mode,

// don't add a block to the chain if it has no let/const variables at all.

return nullptr;

}

return this->Writer()->RecordStartScopeObject(scopeType, scopeLocation, index);

}

// Records the end of the current scope, but only if the current block has block scoped content.

// Otherwise, a scope would not have been added (see ByteCodeGenerator::RecordStartScopeObject()).

void ByteCodeGenerator::RecordEndScopeObject(ParseNode \*pnodeBlock)

{

Assert(pnodeBlock);

if (pnodeBlock->nop == knopBlock && !pnodeBlock->sxBlock.HasBlockScopedContent())

{

return;

}

this->Writer()->RecordEndScopeObject();

}

void BeginEmitBlock(ParseNode \*pnodeBlock, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo)

{

Js::DebuggerScope\* debuggerScope = nullptr;

if (BlockHasOwnScope(pnodeBlock, byteCodeGenerator))

{

Scope \*scope = pnodeBlock->sxBlock.scope;

byteCodeGenerator->PushScope(scope);

Js::RegSlot scopeLocation = scope->GetLocation();

if (scope->GetMustInstantiate())

{

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

scopeLocation = funcInfo->FirstInnerScopeReg() + scope->GetInnerScopeIndex();

if (scope->GetIsObject())

{

debuggerScope = byteCodeGenerator->RecordStartScopeObject(pnodeBlock, Js::DiagExtraScopesType::DiagBlockScopeInObject, scopeLocation);

byteCodeGenerator->Writer()->Unsigned1(Js::OpCode::NewBlockScope, scope->GetInnerScopeIndex());

}

else

{

int scopeIndex = Js::DebuggerScope::InvalidScopeIndex;

debuggerScope = byteCodeGenerator->RecordStartScopeObject(pnodeBlock, Js::DiagExtraScopesType::DiagBlockScopeInSlot, scopeLocation, &scopeIndex);

// TODO: Handle heap enumeration

int scopeSlotCount = scope->GetScopeSlotCount();

byteCodeGenerator->Writer()->Num3(Js::OpCode::NewInnerScopeSlots, scope->GetInnerScopeIndex(), scopeSlotCount + Js::ScopeSlots::FirstSlotIndex, scopeIndex);

}

}

else

{

// In the direct register access case, there is no block scope emitted but we can still track

// the start and end offset of the block. The location registers for let/const variables will still be

// captured along with this range in InitBlockScopedContent().

debuggerScope = byteCodeGenerator->RecordStartScopeObject(pnodeBlock, Js::DiagExtraScopesType::DiagBlockScopeDirect);

}

bool const isGlobalEvalBlockScope = scope->IsGlobalEvalBlockScope();

Js::RegSlot frameDisplayLoc = Js::Constants::NoRegister;

Js::RegSlot tmpInnerEnvReg = Js::Constants::NoRegister;

ParseNodePtr pnodeScope;

for (pnodeScope = pnodeBlock->sxBlock.pnodeScopes; pnodeScope;)

{

switch (pnodeScope->nop)

{

case knopFncDecl:

if (pnodeScope->sxFnc.IsDeclaration())

{

// The frameDisplayLoc register's lifetime has to be controlled by this function. We can't let

// it be released by DefineOneFunction, because further iterations of this loop can allocate

// temps, and we can't let frameDisplayLoc be re-purposed until this loop completes.

// So we'll supply a temp that we allocate and release here.

if (frameDisplayLoc == Js::Constants::NoRegister)

{

if (funcInfo->frameDisplayRegister != Js::Constants::NoRegister)

{

frameDisplayLoc = funcInfo->frameDisplayRegister;

}

else

{

frameDisplayLoc = funcInfo->GetEnvRegister();

}

tmpInnerEnvReg = funcInfo->AcquireTmpRegister();

frameDisplayLoc = byteCodeGenerator->PrependLocalScopes(frameDisplayLoc, tmpInnerEnvReg, funcInfo);

}

byteCodeGenerator->DefineOneFunction(pnodeScope, funcInfo, true, frameDisplayLoc);

}

// If this is the global eval block scope, the function is actually assigned to the global

// so we don't need to keep the registers.

if (isGlobalEvalBlockScope)

{

funcInfo->ReleaseLoc(pnodeScope);

pnodeScope->location = Js::Constants::NoRegister;

}

pnodeScope = pnodeScope->sxFnc.pnodeNext;

break;

case knopBlock:

pnodeScope = pnodeScope->sxBlock.pnodeNext;

break;

case knopCatch:

pnodeScope = pnodeScope->sxCatch.pnodeNext;

break;

case knopWith:

pnodeScope = pnodeScope->sxWith.pnodeNext;

break;

}

}

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

{

funcInfo->ReleaseTmpRegister(tmpInnerEnvReg);

}

if (pnodeBlock->sxBlock.scope->IsGlobalEvalBlockScope() && funcInfo->thisScopeSlot != Js::Constants::NoRegister)

{

Scope\* scope = funcInfo->GetGlobalEvalBlockScope();

byteCodeGenerator->EmitInitCapturedThis(funcInfo, scope);

}

}

else

{

Scope \*scope = pnodeBlock->sxBlock.scope;

if (scope)

{

if (scope->GetMustInstantiate())

{

debuggerScope = byteCodeGenerator->RecordStartScopeObject(pnodeBlock, Js::DiagExtraScopesType::DiagBlockScopeInObject);

}

else

{

debuggerScope = byteCodeGenerator->RecordStartScopeObject(pnodeBlock, Js::DiagExtraScopesType::DiagBlockScopeDirect);

}

}

else

{

debuggerScope = byteCodeGenerator->RecordStartScopeObject(pnodeBlock, Js::DiagExtraScopesType::DiagBlockScopeInSlot);

}

}

byteCodeGenerator->InitBlockScopedContent(pnodeBlock, debuggerScope, funcInfo);

}

void EndEmitBlock(ParseNode \*pnodeBlock, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo)

{

if (BlockHasOwnScope(pnodeBlock, byteCodeGenerator))

{

Scope \*scope = pnodeBlock->sxBlock.scope;

Assert(scope);

Assert(scope == byteCodeGenerator->GetCurrentScope());

byteCodeGenerator->PopScope();

}

byteCodeGenerator->RecordEndScopeObject(pnodeBlock);

}

void CloneEmitBlock(ParseNode \*pnodeBlock, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo)

{

if (BlockHasOwnScope(pnodeBlock, byteCodeGenerator))

{

// Only let variables have observable behavior when there are per iteration

// bindings. const variables do not since they are immutable. Therefore,

// (and the spec agrees), only create new scope clones if the loop variable

// is a let declaration.

bool isConst = false;

pnodeBlock->sxBlock.scope->ForEachSymbolUntil([&isConst](Symbol \* const sym) {

// Exploit the fact that a for loop sxBlock can only have let and const

// declarations, and can only have one or the other, regardless of how

// many syms there might be. Thus only check the first sym.

isConst = sym->GetDecl()->nop == knopConstDecl;

return true;

});

if (!isConst)

{

Scope \*scope = pnodeBlock->sxBlock.scope;

Assert(scope == byteCodeGenerator->GetCurrentScope());

if (scope->GetMustInstantiate())

{

Js::OpCode op = scope->GetIsObject() ? Js::OpCode::CloneBlockScope : Js::OpCode::CloneInnerScopeSlots;

byteCodeGenerator->Writer()->Unsigned1(op, scope->GetInnerScopeIndex());

}

}

}

}

void EmitBlock(ParseNode \*pnodeBlock, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo, BOOL fReturnValue)

{

Assert(pnodeBlock->nop == knopBlock);

ParseNode \*pnode = pnodeBlock->sxBlock.pnodeStmt;

if (pnode == nullptr)

{

return;

}

BeginEmitBlock(pnodeBlock, byteCodeGenerator, funcInfo);

ParseNode \*pnodeLastValStmt = pnodeBlock->sxBlock.pnodeLastValStmt;

while (pnode->nop == knopList)

{

ParseNode\* stmt = pnode->sxBin.pnode1;

if (stmt == pnodeLastValStmt)

{

// This is the last guaranteed return value, so any potential return values have to be

// copied to the return register from this point forward.

pnodeLastValStmt = nullptr;

}

byteCodeGenerator->EmitTopLevelStatement(stmt, funcInfo, fReturnValue && (pnodeLastValStmt == nullptr));

pnode = pnode->sxBin.pnode2;

}

if (pnode == pnodeLastValStmt)

{

pnodeLastValStmt = nullptr;

}

byteCodeGenerator->EmitTopLevelStatement(pnode, funcInfo, fReturnValue && (pnodeLastValStmt == nullptr));

EndEmitBlock(pnodeBlock, byteCodeGenerator, funcInfo);

}

void ClearTmpRegs(ParseNode\* pnode, ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* emitFunc)

{

if (emitFunc->IsTmpReg(pnode->location))

{

pnode->location = Js::Constants::NoRegister;

}

}

void ByteCodeGenerator::EmitTopLevelStatement(ParseNode \*stmt, FuncInfo \*funcInfo, BOOL fReturnValue)

{

if (stmt->nop == knopFncDecl && stmt->sxFnc.IsDeclaration())

{

// Function declarations (not function-declaration RHS's) are already fully processed.

// Skip them here so the temp registers don't get messed up.

return;

}

if (stmt->nop == knopName || stmt->nop == knopDot)

{

// Generating span for top level names are mostly useful in debugging mode, because user can debug it even though no side-effect expected.

// But the name can have runtime error, e.g., foo.bar; // where foo is not defined.

// At this time we need to throw proper line number and offset. so recording on all modes will be useful.

StartStatement(stmt);

Writer()->Empty(Js::OpCode::Nop);

EndStatement(stmt);

}

Emit(stmt, this, funcInfo, fReturnValue);

if (funcInfo->IsTmpReg(stmt->location))

{

if (!stmt->isUsed && !fReturnValue)

{

m\_writer.Reg1(Js::OpCode::Unused, stmt->location);

}

funcInfo->ReleaseLoc(stmt);

}

}

// ByteCodeGenerator::DefineFunctions

//

// Emit byte code for scope-wide function definitions before any calls in the scope, regardless of lexical

// order. Note that stores to the closure array are not emitted until we see the knopFncDecl in the tree

// to make sure that sources of the stores have been defined.

void ByteCodeGenerator::DefineFunctions(FuncInfo \*funcInfoParent)

{

// DefineCachedFunctions doesn't depend on whether the user vars are declared or not, so

// we'll just overload this variable to mean that the functions getting called again and we don't need to do anything

if (funcInfoParent->GetHasCachedScope())

{

this->DefineCachedFunctions(funcInfoParent);

}

else

{

this->DefineUncachedFunctions(funcInfoParent);

}

}

// Iterate over all child functions in a function's parameter and body scopes.

template<typename Fn>

void MapContainerScopeFunctions(ParseNode\* pnodeScope, Fn fn)

{

auto mapFncDeclsInScopeList = [&](ParseNode \*pnodeHead)

{

for (ParseNode \*pnode = pnodeHead; pnode != nullptr;)

{

switch (pnode->nop)

{

case knopFncDecl:

fn(pnode);

pnode = pnode->sxFnc.pnodeNext;

break;

case knopBlock:

pnode = pnode->sxBlock.pnodeNext;

break;

case knopCatch:

pnode = pnode->sxCatch.pnodeNext;

break;

case knopWith:

pnode = pnode->sxWith.pnodeNext;

break;

default:

AssertMsg(false, "Unexpected opcode in tree of scopes");

return;

}

}

};

pnodeScope->sxFnc.MapContainerScopes(mapFncDeclsInScopeList);

}

void ByteCodeGenerator::DefineCachedFunctions(FuncInfo \*funcInfoParent)

{

ParseNode \*pnodeParent = funcInfoParent->root;

uint slotCount = 0;

auto countFncSlots = [&](ParseNode \*pnodeFnc)

{

if (pnodeFnc->sxFnc.GetFuncSymbol() != nullptr && pnodeFnc->sxFnc.IsDeclaration())

{

slotCount++;

}

};

MapContainerScopeFunctions(pnodeParent, countFncSlots);

if (slotCount == 0)

{

return;

}

size\_t extraBytesActual = AllocSizeMath::Mul(slotCount, sizeof(Js::FuncInfoEntry));

// Reg2Aux takes int for byteCount so we need to convert to int. OOM if we can't because it would truncate data.

if (extraBytesActual > INT\_MAX)

{

Js::Throw::OutOfMemory();

}

int extraBytes = (int)extraBytesActual;

Js::FuncInfoArray \*info = AnewPlus(alloc, extraBytes, Js::FuncInfoArray, slotCount);

slotCount = 0;

auto fillEntries = [&](ParseNode \*pnodeFnc)

{

Symbol \*sym = pnodeFnc->sxFnc.GetFuncSymbol();

if (sym != nullptr && (pnodeFnc->sxFnc.IsDeclaration()))

{

AssertMsg(!pnodeFnc->sxFnc.IsGenerator(), "Generator functions are not supported by InitCachedFuncs but since they always escape they should disable function caching");

Js::FuncInfoEntry \*entry = &info->elements[slotCount];

entry->nestedIndex = pnodeFnc->sxFnc.nestedIndex;

entry->scopeSlot = sym->GetScopeSlot();

slotCount++;

}

};

MapContainerScopeFunctions(pnodeParent, fillEntries);

m\_writer.AuxNoReg(Js::OpCode::InitCachedFuncs,

info,

sizeof(Js::FuncInfoArray) + extraBytes,

sizeof(Js::FuncInfoArray) + extraBytes);

slotCount = 0;

auto defineOrGetCachedFunc = [&](ParseNode \*pnodeFnc)

{

Symbol \*sym = pnodeFnc->sxFnc.GetFuncSymbol();

if (pnodeFnc->sxFnc.IsDeclaration())

{

// Do we need to define the function here (i.e., is it not one of our cached locals)?

// Only happens if the sym is null (e.g., function x.y(){}).

if (sym == nullptr)

{

this->DefineOneFunction(pnodeFnc, funcInfoParent);

}

else if (!sym->IsInSlot(funcInfoParent) && sym->GetLocation() != Js::Constants::NoRegister)

{

// If it was defined by InitCachedFuncs, do we need to put it in a register rather than a slot?

m\_writer.Reg1Unsigned1(Js::OpCode::GetCachedFunc, sym->GetLocation(), slotCount);

}

// The "x = function() {...}" case is being generated on the fly, during emission,

// so the caller expects to be able to release this register.

funcInfoParent->ReleaseLoc(pnodeFnc);

pnodeFnc->location = Js::Constants::NoRegister;

slotCount++;

}

};

MapContainerScopeFunctions(pnodeParent, defineOrGetCachedFunc);

AdeletePlus(alloc, extraBytes, info);

}

void ByteCodeGenerator::DefineUncachedFunctions(FuncInfo \*funcInfoParent)

{

ParseNode \*pnodeParent = funcInfoParent->root;

auto defineCheck = [&](ParseNode \*pnodeFnc)

{

Assert(pnodeFnc->nop == knopFncDecl);

//

// Don't define the function upfront in following cases

// 1. x = function() {...};

// Don't define the function for all modes.

// Such a function can only be accessed via the LHS, so we define it at the assignment point

// rather than the scope entry to save a register (and possibly save the whole definition).

//

// 2. x = function f() {...};

// f is not visible in the enclosing scope.

// Such function expressions should be emitted only at the assignment point, as can be used only

// after the assignment. Might save register.

//

if (pnodeFnc->sxFnc.IsDeclaration())

{

this->DefineOneFunction(pnodeFnc, funcInfoParent);

// The "x = function() {...}" case is being generated on the fly, during emission,

// so the caller expects to be able to release this register.

funcInfoParent->ReleaseLoc(pnodeFnc);

pnodeFnc->location = Js::Constants::NoRegister;

}

};

MapContainerScopeFunctions(pnodeParent, defineCheck);

}

void EmitAssignmentToFuncName(ParseNode \*pnodeFnc, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfoParent)

{

// Assign the location holding the func object reference to the given name.

Symbol \*sym = pnodeFnc->sxFnc.pnodeName->sxVar.sym;

if (sym != nullptr && !sym->GetFuncExpr())

{

if (sym->GetIsGlobal())

{

Js::PropertyId propertyId = sym->GetPosition();

byteCodeGenerator->EmitGlobalFncDeclInit(pnodeFnc->location, propertyId, funcInfoParent);

if (byteCodeGenerator->GetScriptContext()->GetConfig()->IsBlockScopeEnabled() &&

byteCodeGenerator->GetFlags() & fscrEval && !funcInfoParent->GetIsStrictMode())

{

byteCodeGenerator->EmitPropStore(pnodeFnc->location, sym, nullptr, funcInfoParent);

}

}

else

{

if (sym->NeedsSlotAlloc(funcInfoParent))

{

if (!sym->GetHasNonCommittedReference() ||

(funcInfoParent->GetParsedFunctionBody()->DoStackNestedFunc()))

{

// No point in trying to optimize if there are no references before we have to commit to slot.

// And not safe to delay putting a stack function in the slot, since we may miss boxing.

sym->SetIsCommittedToSlot();

}

}

byteCodeGenerator->EmitLocalPropInit(pnodeFnc->location, sym, funcInfoParent);

Symbol \* fncScopeSym = sym->GetFuncScopeVarSym();

if (fncScopeSym)

{

Assert(byteCodeGenerator->GetScriptContext()->GetConfig()->IsBlockScopeEnabled());

if (fncScopeSym->GetIsGlobal() && byteCodeGenerator->GetFlags() & fscrEval)

{

Js::PropertyId propertyId = fncScopeSym->GetPosition();

byteCodeGenerator->EmitGlobalFncDeclInit(pnodeFnc->location, propertyId, funcInfoParent);

}

else

{

byteCodeGenerator->EmitPropStore(pnodeFnc->location, fncScopeSym, nullptr, funcInfoParent, false, false, /\* isFncDeclVar \*/true);

}

}

}

}

}

Js::RegSlot ByteCodeGenerator::DefineOneFunction(ParseNode \*pnodeFnc, FuncInfo \*funcInfoParent, bool generateAssignment, Js::RegSlot regEnv, Js::RegSlot frameDisplayTemp)

{

Assert(pnodeFnc->nop == knopFncDecl);

funcInfoParent->AcquireLoc(pnodeFnc);

if (regEnv == Js::Constants::NoRegister)

{

// If the child needs a closure, find a heap-allocated frame to pass to it.

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

{

// We allocated a temp to hold a local frame display value. Use that.

// It's likely that the FD is on the stack, and we used the temp to load it back.

regEnv = frameDisplayTemp;

}

else if (funcInfoParent->frameDisplayRegister != Js::Constants::NoRegister)

{

// This function has built a frame display, so pass it down.

regEnv = funcInfoParent->frameDisplayRegister;

}

else

{

// This function has no captured locals but inherits a closure environment, so pass it down.

regEnv = funcInfoParent->GetEnvRegister();

}

regEnv = this->PrependLocalScopes(regEnv, Js::Constants::NoRegister, funcInfoParent);

}

// AssertMsg(funcInfo->nonLocalSymbols == 0 || regEnv != funcInfoParent->nullConstantRegister,

// "We need a closure for the nested function");

if (regEnv == funcInfoParent->frameDisplayRegister || regEnv == funcInfoParent->GetEnvRegister())

{

m\_writer.NewFunction(pnodeFnc->location, pnodeFnc->sxFnc.nestedIndex, pnodeFnc->sxFnc.IsGenerator());

}

else

{

m\_writer.NewInnerFunction(pnodeFnc->location, pnodeFnc->sxFnc.nestedIndex, regEnv, pnodeFnc->sxFnc.IsGenerator());

}

if (funcInfoParent->IsGlobalFunction() && (this->flags & fscrEval))

{

// A function declared at global scope in eval is untrackable,

// so make sure the caller's cached scope is invalidated.

this->funcEscapes = true;

}

else

{

if (pnodeFnc->sxFnc.IsDeclaration())

{

Symbol \* funcSymbol = pnodeFnc->sxFnc.GetFuncSymbol();

if (funcSymbol)

{

// In the case where a let/const declaration is the same symbol name

// as the function declaration (shadowing case), the let/const var and

// the function declaration symbol are the same and share the same flags

// (particularly, sym->GetIsBlockVar() for this code path).

//

// For example:

// let a = 0; // <-- sym->GetIsBlockVar() = true

// function b(){} // <-- sym2->GetIsBlockVar() = false

//

// let x = 0; // <-- sym3->GetIsBlockVar() = true

// function x(){} // <-- sym3->GetIsBlockVar() = true

//

// In order to tell if the function is actually part

// of a block scope, we compare against the function scope here.

// Note that having a function with the same name as a let/const declaration

// is a redeclaration error, but we're pushing the fix for this out since it's

// a bit involved.

//

// TODO: Once the redeclaration error is in place, this boolean can be replaced

// by funcSymbol->GetIsBlockVar().

Assert(funcInfoParent->GetBodyScope() != nullptr && funcSymbol->GetScope() != nullptr);

bool isFunctionDeclarationInBlock = funcSymbol->GetScope() != funcInfoParent->GetBodyScope();

// Track all vars/lets/consts register slot function declarations.

if (ShouldTrackDebuggerMetadata()

// If this is a let binding function declaration at global level, we want to

// be sure to track the register location as well.

&& !(funcInfoParent->IsGlobalFunction() && !isFunctionDeclarationInBlock))

{

if (!funcSymbol->IsInSlot(funcInfoParent))

{

funcInfoParent->byteCodeFunction->GetFunctionBody()->InsertSymbolToRegSlotList(funcSymbol->GetName(), pnodeFnc->location, funcInfoParent->varRegsCount);

}

}

if (isFunctionDeclarationInBlock)

{

// We only track inner let bindings for the debugger side.

this->TrackFunctionDeclarationPropertyForDebugger(funcSymbol, funcInfoParent);

}

}

}

}

if (pnodeFnc->sxFnc.pnodeName == nullptr || !generateAssignment)

{

return regEnv;

}

EmitAssignmentToFuncName(pnodeFnc, this, funcInfoParent);

return regEnv;

}

void ByteCodeGenerator::DefineUserVars(FuncInfo \*funcInfo)

{

// Initialize scope-wide variables on entry to the scope. TODO: optimize by detecting uses that are always reached

// by an existing initialization.

BOOL fGlobal = funcInfo->IsGlobalFunction();

ParseNode \*pnode;

Js::FunctionBody \*byteCodeFunction = funcInfo->GetParsedFunctionBody();

// Global declarations need a temp register to hold the init value, but the node shouldn't get a register.

// Just assign one on the fly and re-use it for all initializations.

Js::RegSlot tmpReg = fGlobal ? funcInfo->AcquireTmpRegister() : Js::Constants::NoRegister;

for (pnode = funcInfo->root->sxFnc.pnodeVars; pnode; pnode = pnode->sxVar.pnodeNext)

{

Symbol\* sym = pnode->sxVar.sym;

if (sym != nullptr && !(pnode->sxVar.isBlockScopeFncDeclVar && sym->GetIsBlockVar()))

{

if (sym->GetIsCatch() || (pnode->nop == knopVarDecl && sym->GetIsBlockVar()))

{

// The init node was bound to the catch object, because it's inside a catch and has the

// same name as the catch object. But we want to define a user var at function scope,

// so find the right symbol. (We'll still assign the RHS value to the catch object symbol.)

// This also applies to a var declaration in the same scope as a let declaration.

#if DBG

if (!sym->GetIsCatch())

{

// Assert that catch cannot be at function scope and let and var at function scope is redeclaration error.

Assert(funcInfo->bodyScope != sym->GetScope() || !this->scriptContext->GetConfig()->IsBlockScopeEnabled());

}

#endif

sym = funcInfo->bodyScope->FindLocalSymbol(sym->GetName());

Assert(sym && !sym->GetIsCatch() && !sym->GetIsBlockVar());

}

if (sym->GetSymbolType() == STVariable)

{

if (fGlobal)

{

Js::PropertyId propertyId = sym->EnsurePosition(this);

// We do need to initialize some globals to avoid JS errors on loading undefined variables.

// But we first need to make sure we're not trashing built-ins.

if (this->flags & fscrEval)

{

if (funcInfo->byteCodeFunction->GetIsStrictMode())

{

// Check/Init the property of the frame object

this->m\_writer.ElementRootU(Js::OpCode::LdLocalElemUndef,

funcInfo->FindOrAddReferencedPropertyId(propertyId));

}

else

{

// The check and the init involve the first element in the scope chain.

this->m\_writer.ElementScopedU(

Js::OpCode::LdElemUndefScoped, funcInfo->FindOrAddReferencedPropertyId(propertyId));

}

}

else

{

this->m\_writer.ElementU(Js::OpCode::LdElemUndef, ByteCodeGenerator::RootObjectRegister,

funcInfo->FindOrAddReferencedPropertyId(propertyId));

}

}

else if (!sym->GetIsArguments())

{

if (sym->NeedsSlotAlloc(funcInfo))

{

if (!sym->GetHasNonCommittedReference() ||

(sym->GetHasFuncAssignment() && funcInfo->GetParsedFunctionBody()->DoStackNestedFunc()))

{

// No point in trying to optimize if there are no references before we have to commit to slot.

// And not safe to delay putting a stack function in the slot, since we may miss boxing.

sym->SetIsCommittedToSlot();

}

}

if ((!sym->GetHasInit() && !sym->IsInSlot(funcInfo)) ||

(funcInfo->bodyScope->GetIsObject() && !funcInfo->GetHasCachedScope()))

{

Js::RegSlot reg = sym->GetLocation();

if (reg == Js::Constants::NoRegister)

{

Assert(sym->IsInSlot(funcInfo));

reg = funcInfo->AcquireTmpRegister();

}

this->m\_writer.Reg1(Js::OpCode::LdUndef, reg);

this->EmitLocalPropInit(reg, sym, funcInfo);

if (ShouldTrackDebuggerMetadata() && !sym->GetHasInit() && !sym->IsInSlot(funcInfo))

{

byteCodeFunction->InsertSymbolToRegSlotList(sym->GetName(), reg, funcInfo->varRegsCount);

}

funcInfo->ReleaseTmpRegister(reg);

}

}

else if (ShouldTrackDebuggerMetadata())

{

if (!sym->GetHasInit() && !sym->IsInSlot(funcInfo))

{

Js::RegSlot reg = sym->GetLocation();

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

{

byteCodeFunction->InsertSymbolToRegSlotList(sym->GetName(), reg, funcInfo->varRegsCount);

}

}

}

sym->SetHasInit(TRUE);

}

}

}

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

{

funcInfo->ReleaseTmpRegister(tmpReg);

}

for (int i = 0; i < funcInfo->nonUserNonTempRegistersToInitialize.Count(); ++i)

{

m\_writer.Reg1(Js::OpCode::LdUndef, funcInfo->nonUserNonTempRegistersToInitialize.Item(i));

}

this->InitBlockScopedNonTemps(funcInfo->root->sxFnc.pnodeScopes, funcInfo);

}

void ByteCodeGenerator::InitBlockScopedNonTemps(ParseNode \*pnode, FuncInfo \*funcInfo)

{

// Initialize all non-temp register variables on entry to the enclosing func - in particular,

// those with lifetimes that begin after the start of user code and may not be initialized normally.

// This protects us from, for instance, trying to restore garbage on bailout.

// It was originally done in debugger mode only, but we do it always to avoid issues with boxing

// garbage on exit from jitted loop bodies.

while (pnode)

{

switch (pnode->nop)

{

case knopFncDecl:

if (this->scriptContext->GetConfig()->IsBlockScopeEnabled())

{

// If this is a block-scoped function, initialize it.

ParseNode \*pnodeName = pnode->sxFnc.pnodeName;

if (!pnode->sxFnc.IsMethod() && pnodeName && pnodeName->nop == knopVarDecl)

{

Symbol \*sym = pnodeName->sxVar.sym;

Assert(sym);

if (sym->GetLocation() != Js::Constants::NoRegister &&

sym->GetScope()->IsBlockScope(funcInfo) &&

sym->GetScope()->GetFunc() == funcInfo)

{

this->m\_writer.Reg1(Js::OpCode::LdUndef, sym->GetLocation());

}

}

}

// No need to recurse to the nested scopes, as they belong to a nested function.

pnode = pnode->sxFnc.pnodeNext;

break;

case knopBlock:

{

Scope \*scope = pnode->sxBlock.scope;

if (scope)

{

if (scope->IsBlockScope(funcInfo))

{

Js::RegSlot scopeLoc = scope->GetLocation();

if (scopeLoc != Js::Constants::NoRegister && !funcInfo->IsTmpReg(scopeLoc))

{

this->m\_writer.Reg1(Js::OpCode::LdUndef, scopeLoc);

}

}

auto fnInit = [this, funcInfo](ParseNode \*pnode)

{

Symbol \*sym = pnode->sxVar.sym;

if (!sym->IsInSlot(funcInfo) && !sym->GetIsGlobal())

{

this->m\_writer.Reg1(Js::OpCode::InitUndecl, pnode->sxVar.sym->GetLocation());

}

};

IterateBlockScopedVariables(pnode, fnInit);

}

InitBlockScopedNonTemps(pnode->sxBlock.pnodeScopes, funcInfo);

pnode = pnode->sxBlock.pnodeNext;

break;

}

case knopCatch:

InitBlockScopedNonTemps(pnode->sxCatch.pnodeScopes, funcInfo);

pnode = pnode->sxCatch.pnodeNext;

break;

case knopWith:

{

Js::RegSlot withLoc = pnode->location;

AssertMsg(withLoc != Js::Constants::NoRegister && !funcInfo->IsTmpReg(withLoc),

"We should put with objects at known stack locations in debug mode");

this->m\_writer.Reg1(Js::OpCode::LdUndef, withLoc);

InitBlockScopedNonTemps(pnode->sxWith.pnodeScopes, funcInfo);

pnode = pnode->sxWith.pnodeNext;

break;

}

default:

Assert(false);

return;

}

}

}

void ByteCodeGenerator::EmitScopeObjectInit(FuncInfo \*funcInfo)

{

Assert(!funcInfo->byteCodeFunction->GetFunctionBody()->DoStackNestedFunc());

if (!funcInfo->GetHasCachedScope() /\* || forcing scope/inner func caching \*/)

{

return;

}

uint slotCount = funcInfo->bodyScope->GetScopeSlotCount();

uint cachedFuncCount = 0;

Js::PropertyId firstFuncSlot = Js::Constants::NoProperty;

Js::PropertyId firstVarSlot = Js::Constants::NoProperty;

uint extraAlloc = (slotCount + Js::ActivationObjectEx::ExtraSlotCount()) \* sizeof(Js::PropertyId);

// Create and fill the array of local property ID's.

// They all have slots assigned to them already (if they need them): see StartEmitFunction.

Js::PropertyIdArray \*propIds = AnewPlus(alloc, extraAlloc, Js::PropertyIdArray, slotCount);

ParseNode \*pnodeFnc = funcInfo->root;

ParseNode \*pnode;

Symbol \*sym;

if (funcInfo->GetFuncExprNameReference() && pnodeFnc->sxFnc.GetFuncSymbol()->GetScope() == funcInfo->GetBodyScope())

{

Symbol::SaveToPropIdArray(pnodeFnc->sxFnc.GetFuncSymbol(), propIds, this);

}

if (funcInfo->GetHasArguments())

{

// Because the arguments object can access all instances of same-named formals ("function(x,x){...}"),

// be sure we initialize any duplicate appearances of a formal parameter to "NoProperty".

Js::PropertyId slot = 0;

auto initArg = [&](ParseNode \*pnode)

{

if (pnode->IsVarLetOrConst())

{

Symbol \*sym = pnode->sxVar.sym;

Assert(sym);

if (sym->GetScopeSlot() == slot)

{

// This is the last appearance of the formal, so record the ID.

Symbol::SaveToPropIdArray(sym, propIds, this);

}

else

{

// This is an earlier duplicate appearance of the formal, so use NoProperty as a placeholder

// since this slot can't be accessed by name.

Assert(sym->GetScopeSlot() != Js::Constants::NoProperty && sym->GetScopeSlot() > slot);

propIds->elements[slot] = Js::Constants::NoProperty;

}

slot++;

}

};

MapFormalsWithoutRest(pnodeFnc, initArg);

// initArg assumes the sym is in a slot, but this may not be true for the rest parameter.

if (pnodeFnc->sxFnc.pnodeRest != nullptr && pnodeFnc->sxFnc.pnodeRest->sxVar.sym->IsInSlot(funcInfo))

{

initArg(pnodeFnc->sxFnc.pnodeRest);

}

}

else

{

MapFormals(pnodeFnc, [&](ParseNode \*pnode)

{

if (pnode->IsVarLetOrConst())

{

Symbol::SaveToPropIdArray(pnode->sxVar.sym, propIds, this);

}

});

}

auto saveFunctionVarsToPropIdArray = [&](ParseNode \*pnodeFunction)

{

if (pnodeFunction->sxFnc.IsDeclaration())

{

ParseNode \*pnodeName = pnodeFunction->sxFnc.pnodeName;

if (pnodeName != nullptr)

{

while (pnodeName->nop == knopList)

{

if (pnodeName->sxBin.pnode1->nop == knopVarDecl)

{

sym = pnodeName->sxBin.pnode1->sxVar.sym;

if (sym)

{

Symbol::SaveToPropIdArray(sym, propIds, this, &firstFuncSlot);

}

}

pnodeName = pnodeName->sxBin.pnode2;

}

if (pnodeName->nop == knopVarDecl)

{

sym = pnodeName->sxVar.sym;

if (sym)

{

Symbol::SaveToPropIdArray(sym, propIds, this, &firstFuncSlot);

cachedFuncCount++;

}

}

}

}

};

MapContainerScopeFunctions(pnodeFnc, saveFunctionVarsToPropIdArray);

for (pnode = pnodeFnc->sxFnc.pnodeVars; pnode; pnode = pnode->sxVar.pnodeNext)

{

sym = pnode->sxVar.sym;

if (!(pnode->sxVar.isBlockScopeFncDeclVar && sym->GetIsBlockVar()))

{

if (sym->GetIsCatch() || (pnode->nop == knopVarDecl && sym->GetIsBlockVar()))

{

sym = funcInfo->bodyScope->FindLocalSymbol(sym->GetName());

}

Symbol::SaveToPropIdArray(sym, propIds, this, &firstVarSlot);

}

}

ParseNode \*pnodeBlock = pnodeFnc->sxFnc.pnodeScopes;

for (pnode = pnodeBlock->sxBlock.pnodeLexVars; pnode; pnode = pnode->sxVar.pnodeNext)

{

sym = pnode->sxVar.sym;

Symbol::SaveToPropIdArray(sym, propIds, this, &firstVarSlot);

}

pnodeBlock = pnodeFnc->sxFnc.pnodeBodyScope;

for (pnode = pnodeBlock->sxBlock.pnodeLexVars; pnode; pnode = pnode->sxVar.pnodeNext)

{

sym = pnode->sxVar.sym;

Symbol::SaveToPropIdArray(sym, propIds, this, &firstVarSlot);

}

if (funcInfo->thisScopeSlot != Js::Constants::NoRegister)

{

propIds->elements[funcInfo->thisScopeSlot] = Js::PropertyIds::\_lexicalThisSlotSymbol;

}

if (funcInfo->newTargetScopeSlot != Js::Constants::NoRegister)

{

propIds->elements[funcInfo->newTargetScopeSlot] = Js::PropertyIds::\_lexicalNewTargetSymbol;

}

if (funcInfo->superScopeSlot != Js::Constants::NoRegister)

{

propIds->elements[funcInfo->superScopeSlot] = Js::PropertyIds::\_superReferenceSymbol;

}

if (funcInfo->superCtorScopeSlot != Js::Constants::NoRegister)

{

propIds->elements[funcInfo->superCtorScopeSlot] = Js::PropertyIds::\_superCtorReferenceSymbol;

}

// Write the first func slot and first var slot into the auxiliary data

Js::PropertyId \*slots = propIds->elements + slotCount;

slots[0] = cachedFuncCount;

slots[1] = firstFuncSlot;

slots[2] = firstVarSlot;

slots[3] = funcInfo->GetParsedFunctionBody()->NewObjectLiteral();

propIds->hasNonSimpleParams = !funcInfo->root->sxFnc.IsSimpleParameterList();

funcInfo->localPropIdOffset = m\_writer.InsertAuxiliaryData(propIds, sizeof(Js::PropertyIdArray) + extraAlloc);

Assert(funcInfo->localPropIdOffset == 0);

funcInfo->GetParsedFunctionBody()->SetHasCachedScopePropIds(true);

AdeletePlus(alloc, extraAlloc, propIds);

}

void ByteCodeGenerator::FinalizeRegisters(FuncInfo \* funcInfo, Js::FunctionBody \* byteCodeFunction)

{

if (funcInfo->NeedEnvRegister())

{

bool constReg = !funcInfo->GetIsEventHandler() && funcInfo->IsGlobalFunction() && !(this->flags & fscrEval);

funcInfo->AssignEnvRegister(constReg);

}

// Set the function body's constant count before emitting anything so that the byte code writer

// can distinguish constants from variables.

byteCodeFunction->SetConstantCount(funcInfo->constRegsCount);

if (funcInfo->frameDisplayRegister != Js::Constants::NoRegister)

{

byteCodeFunction->SetLocalFrameDisplayReg(funcInfo->frameDisplayRegister);

}

if (funcInfo->frameObjRegister != Js::Constants::NoRegister)

{

byteCodeFunction->SetLocalClosureReg(funcInfo->frameObjRegister);

byteCodeFunction->SetHasScopeObject(true);

}

else if (funcInfo->frameSlotsRegister != Js::Constants::NoRegister)

{

byteCodeFunction->SetLocalClosureReg(funcInfo->frameSlotsRegister);

}

if (this->IsInDebugMode())

{

// Give permanent registers to the inner scopes in debug mode.

uint innerScopeCount = funcInfo->InnerScopeCount();

byteCodeFunction->SetInnerScopeCount(innerScopeCount);

if (innerScopeCount)

{

funcInfo->SetFirstInnerScopeReg(funcInfo->NextVarRegister());

for (uint i = 1; i < innerScopeCount; i++)

{

funcInfo->NextVarRegister();

}

}

}

// NOTE: The FB expects the yield reg to be the final non-temp.

if (byteCodeFunction->IsGenerator())

{

funcInfo->AssignYieldRegister();

}

Js::RegSlot firstTmpReg = funcInfo->varRegsCount;

funcInfo->SetFirstTmpReg(firstTmpReg);

byteCodeFunction->SetFirstTmpReg(funcInfo->RegCount());

}

void ByteCodeGenerator::InitScopeSlotArray(FuncInfo \* funcInfo)

{

// Record slots info for ScopeSlots/ScopeObject.

uint scopeSlotCount = funcInfo->bodyScope->GetScopeSlotCount();

if (scopeSlotCount == 0)

{

return;

}

Js::FunctionBody \*byteCodeFunction = funcInfo->GetParsedFunctionBody();

Js::PropertyId \*propertyIdsForScopeSlotArray = RecyclerNewArrayLeafZ(scriptContext->GetRecycler(), Js::PropertyId, scopeSlotCount);

AssertMsg(!byteCodeFunction->IsReparsed() || byteCodeFunction->m\_wasEverAsmjsMode || byteCodeFunction->scopeSlotArraySize == scopeSlotCount,

"The slot array size is different between debug and non-debug mode");

byteCodeFunction->SetPropertyIdsForScopeSlotArray(propertyIdsForScopeSlotArray, scopeSlotCount);

#if DEBUG

for (UINT i = 0; i < scopeSlotCount; i++)

{

propertyIdsForScopeSlotArray[i] = Js::Constants::NoProperty;

}

#endif

auto setPropIdsForScopeSlotArray = [funcInfo, propertyIdsForScopeSlotArray](Symbol \*const sym)

{

if (sym->NeedsSlotAlloc(funcInfo))

{

// All properties should get correct propertyId here.

Assert(sym->HasScopeSlot()); // We can't allocate scope slot now. Any symbol needing scope slot must have allocated it before this point.

propertyIdsForScopeSlotArray[sym->GetScopeSlot()] = sym->EnsurePosition(funcInfo);

}

};

if (funcInfo->GetParamScope() != nullptr)

{

funcInfo->GetParamScope()->ForEachSymbol(setPropIdsForScopeSlotArray);

}

funcInfo->GetBodyScope()->ForEachSymbol(setPropIdsForScopeSlotArray);

if (funcInfo->thisScopeSlot != Js::Constants::NoRegister)

{

propertyIdsForScopeSlotArray[funcInfo->thisScopeSlot] = Js::PropertyIds::\_lexicalThisSlotSymbol;

}

if (funcInfo->newTargetScopeSlot != Js::Constants::NoRegister)

{

propertyIdsForScopeSlotArray[funcInfo->newTargetScopeSlot] = Js::PropertyIds::\_lexicalNewTargetSymbol;

}

if (funcInfo->superScopeSlot != Js::Constants::NoRegister)

{

propertyIdsForScopeSlotArray[funcInfo->superScopeSlot] = Js::PropertyIds::\_superReferenceSymbol;

}

if (funcInfo->superCtorScopeSlot != Js::Constants::NoRegister)

{

propertyIdsForScopeSlotArray[funcInfo->superCtorScopeSlot] = Js::PropertyIds::\_superCtorReferenceSymbol;

}

#if DEBUG

for (UINT i = 0; i < scopeSlotCount; i++)

{

Assert(propertyIdsForScopeSlotArray[i] != Js::Constants::NoProperty

|| funcInfo->frameObjRegister != Js::Constants::NoRegister); // ScopeObject may have unassigned entries, e.g. for same-named parameters

}

#endif

}

// temporarily load all constants and special registers in a single block

void ByteCodeGenerator::LoadAllConstants(FuncInfo \*funcInfo)

{

Symbol \*sym;

Js::FunctionBody \*byteCodeFunction = funcInfo->GetParsedFunctionBody();

byteCodeFunction->CreateConstantTable();

if (funcInfo->nullConstantRegister != Js::Constants::NoRegister)

{

byteCodeFunction->RecordNullObject(byteCodeFunction->MapRegSlot(funcInfo->nullConstantRegister));

}

if (funcInfo->undefinedConstantRegister != Js::Constants::NoRegister)

{

byteCodeFunction->RecordUndefinedObject(byteCodeFunction->MapRegSlot(funcInfo->undefinedConstantRegister));

}

if (funcInfo->trueConstantRegister != Js::Constants::NoRegister)

{

byteCodeFunction->RecordTrueObject(byteCodeFunction->MapRegSlot(funcInfo->trueConstantRegister));

}

if (funcInfo->falseConstantRegister != Js::Constants::NoRegister)

{

byteCodeFunction->RecordFalseObject(byteCodeFunction->MapRegSlot(funcInfo->falseConstantRegister));

}

if (funcInfo->frameObjRegister != Js::Constants::NoRegister)

{

m\_writer.RecordObjectRegister(funcInfo->frameObjRegister);

if (!funcInfo->GetApplyEnclosesArgs())

{

this->EmitScopeObjectInit(funcInfo);

}

#if DBG

uint count = 0;

funcInfo->GetBodyScope()->ForEachSymbol([&](Symbol \*const sym)

{

if (sym->NeedsSlotAlloc(funcInfo))

{

// All properties should get correct propertyId here.

count++;

}

});

if (funcInfo->GetParamScope() != nullptr)

{

funcInfo->GetParamScope()->ForEachSymbol([&](Symbol \*const sym)

{

if (sym->NeedsSlotAlloc(funcInfo))

{

// All properties should get correct propertyId here.

count++;

}

});

}

// A reparse should result in the same size of the activation object.

// Exclude functions which were created from the ByteCodeCache.

AssertMsg(!byteCodeFunction->IsReparsed() || byteCodeFunction->HasGeneratedFromByteCodeCache() ||

byteCodeFunction->scopeObjectSize == count || byteCodeFunction->m\_wasEverAsmjsMode,

"The activation object size is different between debug and non-debug mode");

byteCodeFunction->scopeObjectSize = count;

#endif

}

else if (funcInfo->frameSlotsRegister != Js::Constants::NoRegister)

{

int scopeSlotCount = funcInfo->bodyScope->GetScopeSlotCount();

if (scopeSlotCount == 0)

{

AssertMsg(funcInfo->frameDisplayRegister != Js::Constants::NoRegister, "Why do we need scope slots?");

m\_writer.Reg1(Js::OpCode::LdC\_A\_Null, funcInfo->frameSlotsRegister);

}

}

if (funcInfo->funcExprScope && funcInfo->funcExprScope->GetIsObject())

{

byteCodeFunction->SetFuncExprScopeReg(funcInfo->funcExprScope->GetLocation());

byteCodeFunction->SetEnvDepth((uint16)-1);

}

bool thisLoadedFromParams = false;

if (funcInfo->NeedEnvRegister())

{

byteCodeFunction->SetEnvReg(funcInfo->GetEnvRegister());

if (funcInfo->GetIsEventHandler())

{

byteCodeFunction->SetThisRegForEventHandler(funcInfo->thisPointerRegister);

// The environment is the namespace hierarchy starting with "this".

Assert(!funcInfo->RegIsConst(funcInfo->GetEnvRegister()));

thisLoadedFromParams = true;

this->InvalidateCachedOuterScopes(funcInfo);

}

else if (funcInfo->IsGlobalFunction() && !(this->flags & fscrEval))

{

Assert(funcInfo->RegIsConst(funcInfo->GetEnvRegister()));

if (funcInfo->GetIsStrictMode())

{

byteCodeFunction->RecordStrictNullDisplayConstant(byteCodeFunction->MapRegSlot(funcInfo->GetEnvRegister()));

}

else

{

byteCodeFunction->RecordNullDisplayConstant(byteCodeFunction->MapRegSlot(funcInfo->GetEnvRegister()));

}

}

else

{

// environment may be required to load "this"

Assert(!funcInfo->RegIsConst(funcInfo->GetEnvRegister()));

this->InvalidateCachedOuterScopes(funcInfo);

}

}

if (funcInfo->frameDisplayRegister != Js::Constants::NoRegister)

{

m\_writer.RecordFrameDisplayRegister(funcInfo->frameDisplayRegister);

}

// new.target may be used to construct the 'this' register so make sure to load it first

if (funcInfo->newTargetRegister != Js::Constants::NoRegister)

{

this->LoadNewTargetObject(funcInfo);

}

if (funcInfo->thisPointerRegister != Js::Constants::NoRegister)

{

this->LoadThisObject(funcInfo, thisLoadedFromParams);

}

this->RecordAllIntConstants(funcInfo);

this->RecordAllStrConstants(funcInfo);

this->RecordAllStringTemplateCallsiteConstants(funcInfo);

funcInfo->doubleConstantToRegister.Map([byteCodeFunction](double d, Js::RegSlot location)

{

byteCodeFunction->RecordFloatConstant(byteCodeFunction->MapRegSlot(location), d);

});

if (funcInfo->GetHasArguments())

{

sym = funcInfo->GetArgumentsSymbol();

Assert(sym);

Assert(funcInfo->GetHasHeapArguments());

if (funcInfo->GetCallsEval() || (!funcInfo->GetApplyEnclosesArgs()))

{

this->LoadHeapArguments(funcInfo);

}

}

else if (!funcInfo->IsGlobalFunction() && !IsInNonDebugMode())

{

uint count = funcInfo->inArgsCount + (funcInfo->root->sxFnc.pnodeRest != nullptr ? 1 : 0) - 1;

if (count != 0)

{

Js::PropertyIdArray \*propIds = RecyclerNewPlus(scriptContext->GetRecycler(), count \* sizeof(Js::PropertyId), Js::PropertyIdArray, count);

GetFormalArgsArray(this, funcInfo, propIds);

byteCodeFunction->SetPropertyIdsOfFormals(propIds);

}

}

//

// If the function is a function expression with a name,

// load the function object at runtime to its activation object.

//

sym = funcInfo->root->sxFnc.GetFuncSymbol();

bool funcExprWithName = !funcInfo->IsGlobalFunction() && sym && sym->GetFuncExpr();

if (funcExprWithName)

{

if (funcInfo->GetFuncExprNameReference() ||

(funcInfo->funcExprScope && funcInfo->funcExprScope->GetIsObject()))

{

//

// x = function f(...) { ... }

// A named function expression's name (Symbol:f) belongs to the enclosing scope.

// Thus there are no uses of 'f' within the scope of the function (as references to 'f'

// are looked up in the closure). So, we can't use f's register as it is from the enclosing

// scope's register namespace. So use a tmp register.

// In ES5 mode though 'f' is \*not\* a part of the enclosing scope. So we always assign 'f' a register

// from it's register namespace, which LdFuncExpr can use.

//

Js::RegSlot ldFuncExprDst = sym->GetLocation();

this->m\_writer.Reg1(Js::OpCode::LdFuncExpr, ldFuncExprDst);

if (sym->IsInSlot(funcInfo))

{

Assert(!this->TopFuncInfo()->GetParsedFunctionBody()->DoStackNestedFunc());

Js::RegSlot scopeLocation;

AnalysisAssert(funcInfo->funcExprScope);

if (funcInfo->funcExprScope->GetIsObject())

{

scopeLocation = funcInfo->funcExprScope->GetLocation();

this->m\_writer.Property(Js::OpCode::StFuncExpr, sym->GetLocation(), scopeLocation,

funcInfo->FindOrAddReferencedPropertyId(sym->GetPosition()));

}

else

{

this->m\_writer.ElementU(Js::OpCode::StLocalFuncExpr, sym->GetLocation(),

funcInfo->FindOrAddReferencedPropertyId(sym->GetPosition()));

}

}

else if (ShouldTrackDebuggerMetadata())

{

funcInfo->byteCodeFunction->GetFunctionBody()->InsertSymbolToRegSlotList(sym->GetName(), sym->GetLocation(), funcInfo->varRegsCount);

}

}

}

}

void ByteCodeGenerator::InvalidateCachedOuterScopes(FuncInfo \*funcInfo)

{

Assert(funcInfo->GetEnvRegister() != Js::Constants::NoRegister);

// Walk the scope stack, from funcInfo outward, looking for scopes that have been cached.

Scope \*scope = funcInfo->GetBodyScope()->GetEnclosingScope();

uint32 envIndex = 0;

while (scope && scope->GetFunc() == funcInfo)

{

// Skip over FuncExpr Scope and parameter scope for current funcInfo to get to the first enclosing scope of the outer function.

scope = scope->GetEnclosingScope();

}

for (; scope; scope = scope->GetEnclosingScope())

{

FuncInfo \*func = scope->GetFunc();

if (scope == func->GetBodyScope())

{

if (func->Escapes() && func->GetHasCachedScope())

{

Assert(scope->GetIsObject());

this->m\_writer.Unsigned1(Js::OpCode::InvalCachedScope, envIndex);

}

}

if (scope->GetMustInstantiate())

{

envIndex++;

}

}

}

void ByteCodeGenerator::LoadThisObject(FuncInfo \*funcInfo, bool thisLoadedFromParams)

{

if (this->scriptContext->GetConfig()->IsES6ClassAndExtendsEnabled() && funcInfo->IsClassConstructor())

{

// Derived class constructors initialize 'this' to be Undecl

// - we'll check this value during a super call and during 'this' access

// Base class constructors initialize 'this' to a new object using new.target

if (funcInfo->IsBaseClassConstructor())

{

EmitBaseClassConstructorThisObject(funcInfo);

}

else

{

this->m\_writer.Reg1(Js::OpCode::InitUndecl, funcInfo->thisPointerRegister);

}

}

else if (!funcInfo->IsGlobalFunction() || (this->flags & fscrEval))

{

//

// thisLoadedFromParams would be true for the event Handler case,

// "this" would have been loaded from parameters to put in the environment

//

if (!thisLoadedFromParams && !funcInfo->IsLambda())

{

m\_writer.ArgIn0(funcInfo->thisPointerRegister);

}

if (!(this->flags & fscrEval) || !funcInfo->IsGlobalFunction())

{

// we don't want to emit 'this' for eval, because 'this' value in eval is equal to 'this' value of caller

// and does not depend on "use strict" inside of eval.

// so we pass 'this' directly in GlobalObject::EntryEval()

EmitThis(funcInfo, funcInfo->thisPointerRegister);

}

}

else

{

Assert(funcInfo->IsGlobalFunction());

Js::RegSlot root = funcInfo->nullConstantRegister;

EmitThis(funcInfo, root);

}

}

void ByteCodeGenerator::LoadNewTargetObject(FuncInfo \*funcInfo)

{

if (funcInfo->IsClassConstructor())

{

Assert(!funcInfo->IsLambda());

m\_writer.ArgIn0(funcInfo->newTargetRegister);

}

else if (funcInfo->IsLambda() && !(this->flags & fscrEval))

{

Scope \*scope;

Js::PropertyId envIndex = -1;

GetEnclosingNonLambdaScope(funcInfo, scope, envIndex);

if (scope->GetFunc()->IsGlobalFunction())

{

m\_writer.Reg1(Js::OpCode::LdUndef, funcInfo->newTargetRegister);

}

else

{

Js::PropertyId slot = scope->GetFunc()->newTargetScopeSlot;

EmitInternalScopedSlotLoad(funcInfo, scope, envIndex, slot, funcInfo->newTargetRegister);

}

}

else if (this->flags & fscrEval)

{

Js::RegSlot scopeLocation;

if (funcInfo->byteCodeFunction->GetIsStrictMode() && funcInfo->IsGlobalFunction())

{

scopeLocation = funcInfo->frameDisplayRegister;

}

else if (funcInfo->NeedEnvRegister())

{

scopeLocation = funcInfo->GetEnvRegister();

}

else

{

// If this eval doesn't have environment register or frame display register, we didn't capture anything from a class constructor.

m\_writer.Reg1(Js::OpCode::LdNewTarget, funcInfo->newTargetRegister);

return;

}

uint cacheId = funcInfo->FindOrAddInlineCacheId(scopeLocation, Js::PropertyIds::\_lexicalNewTargetSymbol, false, false);

this->m\_writer.ElementP(Js::OpCode::ScopedLdFld, funcInfo->newTargetRegister, cacheId);

}

else if (funcInfo->IsGlobalFunction())

{

m\_writer.Reg1(Js::OpCode::LdUndef, funcInfo->newTargetRegister);

}

else

{

m\_writer.Reg1(Js::OpCode::LdNewTarget, funcInfo->newTargetRegister);

}

}

void ByteCodeGenerator::EmitScopeSlotLoadThis(FuncInfo \*funcInfo, Js::RegSlot regLoc, bool chkUndecl)

{

FuncInfo\* nonLambdaFunc = funcInfo;

if (funcInfo->IsLambda())

{

nonLambdaFunc = FindEnclosingNonLambda();

}

if (nonLambdaFunc->IsClassConstructor() && !nonLambdaFunc->IsBaseClassConstructor())

{

// If we are in a derived class constructor and we have a scope slot for 'this',

// we need to load 'this' from the scope slot. This is to support the case where

// the call to initialize 'this' via super() is inside a lambda since the lambda

// can't assign to the 'this' register of the parent constructor.

if (nonLambdaFunc->thisScopeSlot != Js::Constants::NoRegister)

{

Js::PropertyId slot = nonLambdaFunc->thisScopeSlot;

EmitInternalScopedSlotLoad(funcInfo, slot, regLoc, chkUndecl);

}

else if (funcInfo->thisPointerRegister != Js::Constants::NoRegister && chkUndecl)

{

this->m\_writer.Reg1(Js::OpCode::ChkUndecl, funcInfo->thisPointerRegister);

}

else if (chkUndecl)

{

// If we don't have a scope slot for 'this' we know that super could not have

// been called inside a lambda so we can check to see if we called

// super and assigned to the this register already. If not, this should trigger

// a ReferenceError.

EmitUseBeforeDeclarationRuntimeError(this, regLoc, false);

}

}

else if (this->flags & fscrEval && (funcInfo->IsGlobalFunction() || (funcInfo->IsLambda() && nonLambdaFunc->IsGlobalFunction()))

&& funcInfo->GetBodyScope()->GetIsObject())

{

Js::RegSlot scopeLocation;

if (funcInfo->byteCodeFunction->GetIsStrictMode() && funcInfo->IsGlobalFunction())

{

scopeLocation = funcInfo->frameDisplayRegister;

}

else if (funcInfo->NeedEnvRegister())

{

scopeLocation = funcInfo->GetEnvRegister();

}

else

{

// If this eval doesn't have environment register or frame display register, we didn't capture anything from a class constructor

return;

}

// CONSIDER [tawoll] - Should we add a ByteCodeGenerator flag (fscrEvalWithClassConstructorParent) and avoid doing this runtime check?

Js::ByteCodeLabel skipLabel = this->Writer()->DefineLabel();

this->Writer()->BrReg1(Js::OpCode::BrNotUndecl\_A, skipLabel, funcInfo->thisPointerRegister);

uint cacheId = funcInfo->FindOrAddInlineCacheId(scopeLocation, Js::PropertyIds::\_lexicalThisSlotSymbol, false, false);

this->m\_writer.ElementP(Js::OpCode::ScopedLdFld, funcInfo->thisPointerRegister, cacheId);

if (chkUndecl)

{

this->m\_writer.Reg1(Js::OpCode::ChkUndecl, funcInfo->thisPointerRegister);

}

this->Writer()->MarkLabel(skipLabel);

}

}

void ByteCodeGenerator::EmitScopeSlotStoreThis(FuncInfo \*funcInfo, Js::RegSlot regLoc, bool chkUndecl)

{

if (this->flags & fscrEval && (funcInfo->IsGlobalFunction() || (funcInfo->IsLambda() && FindEnclosingNonLambda()->IsGlobalFunction())))

{

Js::RegSlot scopeLocation;

if (funcInfo->byteCodeFunction->GetIsStrictMode() && funcInfo->IsGlobalFunction())

{

scopeLocation = funcInfo->frameDisplayRegister;

}

else

{

scopeLocation = funcInfo->GetEnvRegister();

}

uint cacheId = funcInfo->FindOrAddInlineCacheId(scopeLocation, Js::PropertyIds::\_lexicalThisSlotSymbol, false, true);

this->m\_writer.ElementP(GetScopedStFldOpCode(funcInfo->byteCodeFunction->GetIsStrictMode()), funcInfo->thisPointerRegister, cacheId);

}

else if (regLoc != Js::Constants::NoRegister)

{

EmitInternalScopedSlotStore(funcInfo, regLoc, funcInfo->thisPointerRegister);

}

}

void ByteCodeGenerator::EmitSuperCall(FuncInfo\* funcInfo, ParseNode\* pnode, BOOL fReturnValue)

{

Assert(pnode->sxCall.pnodeTarget->nop == knopSuper);

FuncInfo\* nonLambdaFunc = funcInfo;

if (funcInfo->IsLambda())

{

nonLambdaFunc = this->FindEnclosingNonLambda();

}

if (nonLambdaFunc->IsBaseClassConstructor())

{

// super() is not allowed in base class constructors. If we detect this, emit a ReferenceError and skip making the call.

this->Writer()->W1(Js::OpCode::RuntimeReferenceError, SCODE\_CODE(JSERR\_ClassSuperInBaseClass));

return;

}

else

{

EmitSuperFieldPatch(funcInfo, pnode, this);

pnode->isUsed = true;

}

// We already know pnode->sxCall.pnodeTarget->nop is super but we can't use the super register in case

// this is an eval and we will load super dynamically from the scope using ScopedLdSuper.

// That means we'll have to rely on the location of the call target to be sure.

// We have to make sure to allocate the location for the node now, before we try to branch on it.

Emit(pnode->sxCall.pnodeTarget, this, funcInfo, false, /\*isConstructorCall\*/ true); // reuse isConstructorCall ("new super()" is illegal)

//

// if (super is class constructor) {

// \_this = new.target;

// } else {

// \_this = NewScObjFull(new.target);

// }

//

// temp = super.call(\_this, new.target); // CallFlag\_New | CallFlag\_NewTarget | CallFlag\_ExtraArg

// if (temp is object) {

// \_this = temp;

// }

//

// if (UndeclBlockVar === this) {

// this = \_this;

// } else {

// throw ReferenceError;

// }

//

funcInfo->AcquireLoc(pnode);

Js::RegSlot thisForSuperCall = funcInfo->AcquireTmpRegister();

Js::ByteCodeLabel useNewTargetForThisLabel = this->Writer()->DefineLabel();

Js::ByteCodeLabel makeCallLabel = this->Writer()->DefineLabel();

Js::ByteCodeLabel useSuperCallResultLabel = this->Writer()->DefineLabel();

Js::ByteCodeLabel doneLabel = this->Writer()->DefineLabel();

this->Writer()->BrReg1(Js::OpCode::BrOnClassConstructor, useNewTargetForThisLabel, pnode->sxCall.pnodeTarget->location);

this->Writer()->Reg2(Js::OpCode::NewScObjectNoCtorFull, thisForSuperCall, funcInfo->newTargetRegister);

this->Writer()->Br(Js::OpCode::Br, makeCallLabel);

this->Writer()->MarkLabel(useNewTargetForThisLabel);

this->Writer()->Reg2(Js::OpCode::Ld\_A, thisForSuperCall, funcInfo->newTargetRegister);

this->Writer()->MarkLabel(makeCallLabel);

EmitCall(pnode, Js::Constants::NoRegister, this, funcInfo, fReturnValue, /\*fEvaluateComponents\*/ true, /\*fHasNewTarget\*/ true, thisForSuperCall);

// We have to use another temp for the this value before assigning to this register.

// This is because IRBuilder does not expect us to use the value of a temp after potentially assigning to that same temp.

// Ex:

// \_this = new.target;

// temp = super.call(\_this);

// if (temp is object) {

// \_this = temp; // creates a new sym for \_this as it was previously used

// }

// this = \_this; // tries to loads a value from the old sym (which is dead)

Js::RegSlot valueForThis = funcInfo->AcquireTmpRegister();

this->Writer()->BrReg1(Js::OpCode::BrOnObject\_A, useSuperCallResultLabel, pnode->location);

this->Writer()->Reg2(Js::OpCode::Ld\_A, valueForThis, thisForSuperCall);

this->Writer()->Br(Js::OpCode::Br, doneLabel);

this->Writer()->MarkLabel(useSuperCallResultLabel);

this->Writer()->Reg2(Js::OpCode::Ld\_A, valueForThis, pnode->location);

this->Writer()->MarkLabel(doneLabel);

// The call is done and we know what we will bind to 'this' so let's check to see if 'this' is already decl.

// We may need to load 'this' from the scope slot.

EmitScopeSlotLoadThis(funcInfo, funcInfo->thisPointerRegister, false);

Js::ByteCodeLabel skipLabel = this->Writer()->DefineLabel();

Js::RegSlot tmpUndeclReg = funcInfo->AcquireTmpRegister();

this->Writer()->Reg1(Js::OpCode::InitUndecl, tmpUndeclReg);

this->Writer()->BrReg2(Js::OpCode::BrSrEq\_A, skipLabel, funcInfo->thisPointerRegister, tmpUndeclReg);

funcInfo->ReleaseTmpRegister(tmpUndeclReg);

this->Writer()->W1(Js::OpCode::RuntimeReferenceError, SCODE\_CODE(JSERR\_ClassThisAlreadyAssigned));

this->Writer()->MarkLabel(skipLabel);

this->Writer()->Reg2(Js::OpCode::StrictLdThis, funcInfo->thisPointerRegister, valueForThis);

funcInfo->ReleaseTmpRegister(valueForThis);

funcInfo->ReleaseTmpRegister(thisForSuperCall);

// We already assigned the result of super() to the 'this' register but we need to store it in the scope slot, too. If there is one.

this->EmitScopeSlotStoreThis(funcInfo, nonLambdaFunc->thisScopeSlot);

}

void ByteCodeGenerator::EmitClassConstructorEndCode(FuncInfo \*funcInfo)

{

if (funcInfo->thisPointerRegister != Js::Constants::NoRegister)

{

// We need to try and load 'this' from the scope slot, if there is one.

EmitScopeSlotLoadThis(funcInfo, funcInfo->thisPointerRegister);

this->Writer()->Reg2(Js::OpCode::Ld\_A, ByteCodeGenerator::ReturnRegister, funcInfo->thisPointerRegister);

}

else

{

// This is the case where we don't have any references to this or super in the constructor or any nested lambda functions.

// We know 'this' must be undecl so let's just emit the ReferenceError as part of the fallthrough.

EmitUseBeforeDeclarationRuntimeError(this, ByteCodeGenerator::ReturnRegister, true);

}

}

void ByteCodeGenerator::EmitBaseClassConstructorThisObject(FuncInfo \*funcInfo)

{

this->Writer()->Reg2(Js::OpCode::NewScObjectNoCtorFull, funcInfo->thisPointerRegister, funcInfo->newTargetRegister);

}

void ByteCodeGenerator::EmitInternalScopedSlotLoad(FuncInfo \*funcInfo, Js::RegSlot slot, Js::RegSlot symbolRegister, bool chkUndecl)

{

Scope\* scope = nullptr;

if (funcInfo->IsLambda())

{

Js::PropertyId envIndex = -1;

GetEnclosingNonLambdaScope(funcInfo, scope, envIndex);

EmitInternalScopedSlotLoad(funcInfo, scope, envIndex, slot, symbolRegister, chkUndecl);

}

else

{

scope = funcInfo->GetBodyScope();

EmitInternalScopedSlotLoad(funcInfo, scope, -1, slot, symbolRegister, chkUndecl);

}

}

void ByteCodeGenerator::EmitInternalScopedSlotLoad(FuncInfo \*funcInfo, Scope \*scope, Js::PropertyId envIndex, Js::RegSlot slot, Js::RegSlot symbolRegister, bool chkUndecl)

{

Assert(slot != Js::Constants::NoProperty);

Js::ProfileId profileId = funcInfo->FindOrAddSlotProfileId(scope, symbolRegister);

Js::OpCode opcode;

Js::RegSlot scopeLocation = scope->GetLocation();

opcode = this->GetLdSlotOp(scope, envIndex, scopeLocation, funcInfo);

slot += (scope->GetIsObject() ? 0 : Js::ScopeSlots::FirstSlotIndex);

if (envIndex != -1)

{

this->m\_writer.SlotI2(opcode, symbolRegister, envIndex + Js::FrameDisplay::GetOffsetOfScopes()/sizeof(Js::Var), slot, profileId);

}

else if (scopeLocation != Js::Constants::NoRegister &&

(scopeLocation == funcInfo->frameSlotsRegister || scopeLocation == funcInfo->frameObjRegister))

{

this->m\_writer.SlotI1(opcode, symbolRegister, slot, profileId);

}

else

{

this->m\_writer.Slot(opcode, symbolRegister, scopeLocation, slot, profileId);

}

if (chkUndecl)

{

this->m\_writer.Reg1(Js::OpCode::ChkUndecl, symbolRegister);

}

}

void ByteCodeGenerator::EmitInternalScopedSlotStore(FuncInfo \*funcInfo, Js::RegSlot slot, Js::RegSlot symbolRegister)

{

Assert(slot != Js::Constants::NoProperty);

Scope\* scope = nullptr;

Js::OpCode opcode;

Js::PropertyId envIndex = -1;

if (funcInfo->IsLambda())

{

GetEnclosingNonLambdaScope(funcInfo, scope, envIndex);

}

else

{

scope = funcInfo->GetBodyScope();

}

Js::RegSlot scopeLocation = scope->GetLocation();

opcode = this->GetStSlotOp(scope, envIndex, scopeLocation, false, funcInfo);

slot += (scope->GetIsObject() ? 0 : Js::ScopeSlots::FirstSlotIndex);

if (envIndex != -1)

{

this->m\_writer.SlotI2(opcode, symbolRegister, envIndex + Js::FrameDisplay::GetOffsetOfScopes()/sizeof(Js::Var), slot);

}

else if (scopeLocation != Js::Constants::NoRegister &&

(scopeLocation == funcInfo->frameSlotsRegister || scopeLocation == funcInfo->frameObjRegister))

{

this->m\_writer.SlotI1(opcode, symbolRegister, slot);

}

else if (scope->GetIsObject())

{

this->m\_writer.Slot(opcode, symbolRegister, scopeLocation, slot);

}

else

{

this->m\_writer.SlotI2(opcode, symbolRegister, scope->GetInnerScopeIndex(), slot);

}

}

void ByteCodeGenerator::EmitInternalScopeObjInit(FuncInfo \*funcInfo, Scope \*scope, Js::RegSlot valueLocation, Js::PropertyId propertyId)

{

Js::RegSlot scopeLocation = scope->GetLocation();

Js::OpCode opcode = this->GetInitFldOp(scope, scopeLocation, funcInfo);

if (scopeLocation != Js::Constants::NoRegister && scopeLocation == funcInfo->frameObjRegister)

{

uint cacheId = funcInfo->FindOrAddInlineCacheId(scopeLocation, propertyId, false, true);

this->m\_writer.ElementP(opcode, valueLocation, cacheId);

}

else if (scope->HasInnerScopeIndex())

{

uint cacheId = funcInfo->FindOrAddInlineCacheId(funcInfo->InnerScopeToRegSlot(scope), propertyId, false, true);

this->m\_writer.ElementPIndexed(opcode, valueLocation, scope->GetInnerScopeIndex(), cacheId);

}

else

{

uint cacheId = funcInfo->FindOrAddInlineCacheId(scopeLocation, propertyId, false, true);

this->m\_writer.PatchableProperty(opcode, valueLocation, scopeLocation, cacheId);

}

}

void ByteCodeGenerator::GetEnclosingNonLambdaScope(FuncInfo \*funcInfo, Scope \* &scope, Js::PropertyId &envIndex)

{

envIndex = -1;

for (scope = funcInfo->GetBodyScope()->GetEnclosingScope(); scope; scope = scope->GetEnclosingScope())

{

if (scope->GetMustInstantiate())

{

envIndex++;

}

if ((scope == scope->GetFunc()->GetBodyScope() && !scope->GetFunc()->IsLambda()) || scope->IsGlobalEvalBlockScope())

{

break;

}

}

}

void ByteCodeGenerator::EmitThis(FuncInfo \*funcInfo, Js::RegSlot fromRegister)

{

if (funcInfo->IsLambda())

{

Scope \*scope;

Js::PropertyId envIndex = -1;

GetEnclosingNonLambdaScope(funcInfo, scope, envIndex);

FuncInfo\* parent = scope->GetFunc();

if (parent->IsGlobalFunction())

{

if (this->flags & fscrEval)

{

scope = parent->GetGlobalEvalBlockScope();

Js::PropertyId slot = parent->thisScopeSlot;

EmitInternalScopedSlotLoad(funcInfo, scope, envIndex, slot, funcInfo->thisPointerRegister, false);

}

else

{

// Always load global object via LdThis of null to get the possibly protected via secureHostObject global object.

this->m\_writer.Reg2Int1(Js::OpCode::LdThis, funcInfo->thisPointerRegister, funcInfo->nullConstantRegister, this->GetModuleID());

}

}

else if (!parent->IsClassConstructor() || parent->IsBaseClassConstructor())

{

// In a lambda inside a derived class constructor, 'this' should be loaded from the scope slot whenever 'this' is accessed.

// It's safe to load 'this' into the register for base class constructors because there is no complex assignment to 'this'

// via super call chain.

Js::PropertyId slot = parent->thisScopeSlot;

EmitInternalScopedSlotLoad(funcInfo, scope, envIndex, slot, funcInfo->thisPointerRegister, false);

}

}

else if (funcInfo->byteCodeFunction->GetIsStrictMode() && (!funcInfo->IsGlobalFunction() || this->flags & fscrEval))

{

m\_writer.Reg2(Js::OpCode::StrictLdThis, funcInfo->thisPointerRegister, fromRegister);

}

else

{

m\_writer.Reg2Int1(Js::OpCode::LdThis, funcInfo->thisPointerRegister, fromRegister, this->GetModuleID());

}

}

void ByteCodeGenerator::EmitLoadFormalIntoRegister(ParseNode \*pnodeFormal, Js::RegSlot pos, FuncInfo \*funcInfo)

{

if (pnodeFormal->IsVarLetOrConst())

{

// Get the param from its argument position into its assigned register.

// The position should match the location, otherwise, it has been shadowed by parameter with the same name

Symbol \*formal = pnodeFormal->sxVar.sym;

if (formal->GetLocation() + 1 == pos)

{

// Transfer to the frame object, etc., if necessary.

this->EmitLocalPropInit(formal->GetLocation(), formal, funcInfo);

}

if (ShouldTrackDebuggerMetadata() && !formal->IsInSlot(funcInfo))

{

Assert(!formal->GetHasInit());

funcInfo->GetParsedFunctionBody()->InsertSymbolToRegSlotList(formal->GetName(), formal->GetLocation(), funcInfo->varRegsCount);

}

}

}

void ByteCodeGenerator::HomeArguments(FuncInfo \*funcInfo)

{

// Transfer formal parameters to their home locations on the local frame.

if (funcInfo->GetHasArguments())

{

if (funcInfo->root->sxFnc.pnodeRest != nullptr)

{

// Since we don't have to iterate over arguments here, we'll trust the location to be correct.

EmitLoadFormalIntoRegister(funcInfo->root->sxFnc.pnodeRest, funcInfo->root->sxFnc.pnodeRest->sxVar.sym->GetLocation() + 1, funcInfo);

}

// The arguments object creation helper does this work for us.

return;

}

Js::ArgSlot pos = 1;

auto loadFormal = [&](ParseNode \*pnodeFormal)

{

EmitLoadFormalIntoRegister(pnodeFormal, pos, funcInfo);

pos++;

};

MapFormals(funcInfo->root, loadFormal);

}

void ByteCodeGenerator::DefineLabels(FuncInfo \*funcInfo)

{

funcInfo->singleExit = m\_writer.DefineLabel();

SList<ParseNode \*>::Iterator iter(&funcInfo->targetStatements);

while (iter.Next())

{

ParseNode \* node = iter.Data();

node->sxStmt.breakLabel = m\_writer.DefineLabel();

node->sxStmt.continueLabel = m\_writer.DefineLabel();

node->emitLabels = true;

}

}

void ByteCodeGenerator::EmitGlobalBody(FuncInfo \*funcInfo)

{

// Emit global code (global scope or eval), fixing up the return register with the implicit

// return value.

ParseNode \*pnode = funcInfo->root->sxFnc.pnodeBody;

ParseNode \*pnodeLastVal = funcInfo->root->sxProg.pnodeLastValStmt;

if (pnodeLastVal == nullptr)

{

// We're not guaranteed to compute any values, so fix up the return register at the top

// in case.

this->m\_writer.Reg1(Js::OpCode::LdUndef, ReturnRegister);

}

while (pnode->nop == knopList)

{

ParseNode \*stmt = pnode->sxBin.pnode1;

if (stmt == pnodeLastVal)

{

pnodeLastVal = nullptr;

}

if (pnodeLastVal == nullptr && (this->flags & fscrReturnExpression))

{

EmitTopLevelStatement(stmt, funcInfo, true);

}

else

{

// Haven't hit the post-dominating return value yet,

// so don't bother with the return register.

EmitTopLevelStatement(stmt, funcInfo, false);

}

pnode = pnode->sxBin.pnode2;

}

EmitTopLevelStatement(pnode, funcInfo, false);

}

void ByteCodeGenerator::EmitFunctionBody(FuncInfo \*funcInfo)

{

// Emit a function body. Only explicit returns and the implicit "undef" at the bottom

// get copied to the return register.

ParseNode \*pnodeBody = funcInfo->root->sxFnc.pnodeBody;

ParseNode \*pnode = pnodeBody;

while (pnode->nop == knopList)

{

ParseNode \*stmt = pnode->sxBin.pnode1;

if (stmt->CapturesSyms())

{

CapturedSymMap \*map = funcInfo->EnsureCapturedSymMap();

SList<Symbol\*> \*list = map->Item(stmt);

FOREACH\_SLIST\_ENTRY(Symbol\*, sym, list)

{

if (!sym->GetIsCommittedToSlot())

{

Assert(sym->GetLocation() != Js::Constants::NoProperty);

sym->SetIsCommittedToSlot();

ParseNode \*decl = sym->GetDecl();

Assert(decl);

if (PHASE\_TRACE(Js::DelayCapturePhase, funcInfo->byteCodeFunction))

{

Output::Print(L"--- DelayCapture: Committed symbol '%s' to slot.\n", sym->GetName());

Output::Flush();

}

this->EmitPropStore(sym->GetLocation(), sym, sym->GetPid(), funcInfo, decl->nop == knopLetDecl, decl->nop == knopConstDecl);

}

}

NEXT\_SLIST\_ENTRY;

}

EmitTopLevelStatement(stmt, funcInfo, false);

pnode = pnode->sxBin.pnode2;

}

Assert(!pnode->CapturesSyms());

EmitTopLevelStatement(pnode, funcInfo, false);

}

void ByteCodeGenerator::EmitProgram(ParseNode \*pnodeProg)

{

// Indicate that the binding phase is over.

this->isBinding = false;

this->trackEnvDepth = true;

AssignPropertyIds(pnodeProg->sxFnc.funcInfo->byteCodeFunction);

long initSize = this->maxAstSize / AstBytecodeRatioEstimate;

// Use the temp allocator in bytecode write temp buffer.

m\_writer.InitData(this->alloc, initSize);

#ifdef LOG\_BYTECODE\_AST\_RATIO

// log the max Ast size

Output::Print(L"Max Ast size: %d", initSize);

#endif

Assert(pnodeProg && pnodeProg->nop == knopProg);

if (this->parentScopeInfo)

{

// Scope stack is already set up the way we want it, so don't visit the global scope.

// Start emitting with the nested scope (i.e., the deferred function).

this->EmitScopeList(pnodeProg->sxProg.pnodeScopes);

}

else

{

this->EmitScopeList(pnodeProg);

}

}

void ByteCodeGenerator::EmitInitCapturedThis(FuncInfo\* funcInfo, Scope\* scope)

{

if (scope->GetIsObject())

{

// Ensure space for the this slot

this->EmitInternalScopeObjInit(funcInfo, scope, funcInfo->thisPointerRegister, Js::PropertyIds::\_lexicalThisSlotSymbol);

}

else

{

this->EmitInternalScopedSlotStore(funcInfo, funcInfo->thisScopeSlot, funcInfo->thisPointerRegister);

}

}

void ByteCodeGenerator::EmitInitCapturedNewTarget(FuncInfo\* funcInfo, Scope\* scope)

{

if (scope->GetIsObject())

{

// Ensure space for the new.target slot

this->EmitInternalScopeObjInit(funcInfo, scope, funcInfo->newTargetRegister, Js::PropertyIds::\_lexicalNewTargetSymbol);

}

else

{

this->EmitInternalScopedSlotStore(funcInfo, funcInfo->newTargetScopeSlot, funcInfo->newTargetRegister);

}

}

void EmitDestructuredObject(ParseNode \*lhs, Js::RegSlot rhsLocation, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo);

void EmitDestructuredValueOrInitializer(ParseNodePtr lhsElementNode, Js::RegSlot rhsLocation, ParseNodePtr initializer, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo);

void ByteCodeGenerator::EmitDefaultArgs(FuncInfo \*funcInfo, ParseNode \*pnode)

{

auto emitDefaultArg = [&](ParseNode \*pnodeArg)

{

if (pnodeArg->nop == knopParamPattern)

{

this->StartStatement(pnodeArg);

Assert(pnodeArg->sxParamPattern.location != Js::Constants::NoRegister);

ParseNodePtr pnode1 = pnodeArg->sxParamPattern.pnode1;

if (pnode1->IsPattern())

{

EmitAssignment(nullptr, pnode1, pnodeArg->sxParamPattern.location, this, funcInfo);

}

else

{

Assert(pnode1->nop == knopAsg);

Assert(pnode1->sxBin.pnode1->IsPattern());

EmitDestructuredValueOrInitializer(pnode1->sxBin.pnode1, pnodeArg->sxParamPattern.location, pnode1->sxBin.pnode2, this, funcInfo);

}

this->EndStatement(pnodeArg);

return;

}

else if (pnodeArg->IsVarLetOrConst())

{

Js::RegSlot location = pnodeArg->sxVar.sym->GetLocation();

if (pnodeArg->sxVar.pnodeInit == nullptr)

{

// Since the formal hasn't been initialized in LdLetHeapArguments, we'll initialize it here.

EmitPropStore(location, pnodeArg->sxVar.sym, pnodeArg->sxVar.pid, funcInfo, true);

pnodeArg->sxVar.sym->SetNeedDeclaration(false);

return;

}

// Load the default argument if we got undefined, skip RHS evaluation otherwise.

Js::ByteCodeLabel noDefaultLabel = this->m\_writer.DefineLabel();

Js::ByteCodeLabel endLabel = this->m\_writer.DefineLabel();

this->StartStatement(pnodeArg);

m\_writer.BrReg2(Js::OpCode::BrNeq\_A, noDefaultLabel, location, funcInfo->undefinedConstantRegister);

Emit(pnodeArg->sxVar.pnodeInit, this, funcInfo, false);

pnodeArg->sxVar.sym->SetNeedDeclaration(false); // After emit to prevent foo(a = a)

if (funcInfo->GetHasArguments() && pnodeArg->sxVar.sym->IsInSlot(funcInfo))

{

EmitPropStore(pnodeArg->sxVar.pnodeInit->location, pnodeArg->sxVar.sym, pnodeArg->sxVar.pid, funcInfo, true);

m\_writer.Br(endLabel);

}

else

{

EmitAssignment(nullptr, pnodeArg, pnodeArg->sxVar.pnodeInit->location, this, funcInfo);

}

funcInfo->ReleaseLoc(pnodeArg->sxVar.pnodeInit);

m\_writer.MarkLabel(noDefaultLabel);

if (funcInfo->GetHasArguments() && pnodeArg->sxVar.sym->IsInSlot(funcInfo))

{

EmitPropStore(location, pnodeArg->sxVar.sym, pnodeArg->sxVar.pid, funcInfo, true);

m\_writer.MarkLabel(endLabel);

}

this->EndStatement(pnodeArg);

}

};

// If the function is async, we wrap the default arguments in a try catch and reject a Promise in case of error.

if (pnode->sxFnc.IsAsync())

{

uint cacheId;

Js::ByteCodeLabel catchLabel = m\_writer.DefineLabel();

Js::ByteCodeLabel doneLabel = m\_writer.DefineLabel();

Js::RegSlot catchArgLocation = funcInfo->AcquireTmpRegister();

Js::RegSlot promiseLocation = funcInfo->AcquireTmpRegister();

Js::RegSlot rejectLocation = funcInfo->AcquireTmpRegister();

// try

m\_writer.RecordCrossFrameEntryExitRecord(/\* isEnterBlock = \*/ true);

m\_writer.Br(Js::OpCode::TryCatch, catchLabel);

// Rest cannot have a default argument, so we ignore it.

MapFormalsWithoutRest(pnode, emitDefaultArg);

m\_writer.RecordCrossFrameEntryExitRecord(/\* isEnterBlock = \*/ false);

m\_writer.Empty(Js::OpCode::Leave);

m\_writer.Br(doneLabel);

// catch

m\_writer.MarkLabel(catchLabel);

m\_writer.Reg1(Js::OpCode::Catch, catchArgLocation);

m\_writer.RecordCrossFrameEntryExitRecord(/\* isEnterBlock = \*/ true);

m\_writer.Empty(Js::OpCode::Nop);

// return Promise.reject(error);

cacheId = funcInfo->FindOrAddRootObjectInlineCacheId(Js::PropertyIds::Promise, false, false);

m\_writer.PatchableRootProperty(Js::OpCode::LdRootFld, promiseLocation, cacheId, false, false);

EmitInvoke(rejectLocation, promiseLocation, Js::PropertyIds::reject, this, funcInfo, catchArgLocation);

m\_writer.Reg2(Js::OpCode::Ld\_A, ByteCodeGenerator::ReturnRegister, rejectLocation);

m\_writer.RecordCrossFrameEntryExitRecord(/\* isEnterBlock = \*/ false);

m\_writer.Empty(Js::OpCode::Leave);

m\_writer.Br(funcInfo->singleExit);

m\_writer.Empty(Js::OpCode::Leave);

m\_writer.MarkLabel(doneLabel);

this->SetHasTry(true);

funcInfo->ReleaseTmpRegister(rejectLocation);

funcInfo->ReleaseTmpRegister(promiseLocation);

funcInfo->ReleaseTmpRegister(catchArgLocation);

}

else

{

// Rest cannot have a default argument, so we ignore it.

MapFormalsWithoutRest(pnode, emitDefaultArg);

}

}

void ByteCodeGenerator::EmitOneFunction(ParseNode \*pnode)

{

Assert(pnode && (pnode->nop == knopProg || pnode->nop == knopFncDecl));

FuncInfo \*funcInfo = pnode->sxFnc.funcInfo;

Assert(funcInfo != nullptr);

if (funcInfo->IsFakeGlobalFunction(this->flags))

{

return;

}

Js::ParseableFunctionInfo\* deferParseFunction = funcInfo->byteCodeFunction;

deferParseFunction->SetGrfscr(deferParseFunction->GetGrfscr() | (this->flags & ~fscrDeferredFncExpression));

deferParseFunction->SetSourceInfo(this->GetCurrentSourceIndex(),

funcInfo->root,

!!(this->flags & fscrEvalCode),

((this->flags & fscrDynamicCode) && !(this->flags & fscrEvalCode)));

deferParseFunction->SetInParamsCount(funcInfo->inArgsCount);

if (pnode->sxFnc.HasDefaultArguments())

{

deferParseFunction->SetReportedInParamsCount(pnode->sxFnc.firstDefaultArg + 1);

}

else

{

deferParseFunction->SetReportedInParamsCount(funcInfo->inArgsCount);

}

if (funcInfo->root->sxFnc.pnodeBody == nullptr)

{

if (scriptContext->GetConfig()->BindDeferredPidRefs() &&

!PHASE\_OFF1(Js::SkipNestedDeferredPhase))

{

deferParseFunction->BuildDeferredStubs(funcInfo->root);

}

return;

}

Js::FunctionBody\* byteCodeFunction = funcInfo->GetParsedFunctionBody();

// We've now done a full parse of this function, so we no longer need to remember the extents

// and attributes of the top-level nested functions. (The above code has run for all of those,

// so they have pointers to the stub sub-trees they need.)

byteCodeFunction->SetDeferredStubs(nullptr);

try

{

// Bug : 301517

// In the debug mode the hasOnlyThis optimization needs to be disabled, since user can break in this function

// and do operation on 'this' and its property, which may not be defined yet.

if (funcInfo->root->sxFnc.HasOnlyThisStmts() && !IsInDebugMode())

{

byteCodeFunction->SetHasOnlyThisStmts(true);

}

if (byteCodeFunction->IsInlineApplyDisabled() || this->scriptContext->GetConfig()->IsNoNative())

{

if ((pnode->nop == knopFncDecl) && (funcInfo->GetHasHeapArguments()) && (!funcInfo->GetCallsEval()) && ApplyEnclosesArgs(pnode, this))

{

bool applyEnclosesArgs = true;

for (ParseNode\* pnodeVar = funcInfo->root->sxFnc.pnodeVars; pnodeVar; pnodeVar = pnodeVar->sxVar.pnodeNext)

{

Symbol\* sym = pnodeVar->sxVar.sym;

if (sym->GetSymbolType() == STVariable && !sym->GetIsArguments())

{

applyEnclosesArgs = false;

break;

}

}

auto constAndLetCheck = [](ParseNode \*pnodeBlock, bool \*applyEnclosesArgs)

{

if (\*applyEnclosesArgs)

{

for (auto lexvar = pnodeBlock->sxBlock.pnodeLexVars; lexvar; lexvar = lexvar->sxVar.pnodeNext)

{

Symbol\* sym = lexvar->sxVar.sym;

if (sym->GetSymbolType() == STVariable && !sym->GetIsArguments())

{

\*applyEnclosesArgs = false;

break;

}

}

}

};

constAndLetCheck(funcInfo->root->sxFnc.pnodeScopes, &applyEnclosesArgs);

constAndLetCheck(funcInfo->root->sxFnc.pnodeBodyScope, &applyEnclosesArgs);

funcInfo->SetApplyEnclosesArgs(applyEnclosesArgs);

}

}

if (!funcInfo->IsGlobalFunction())

{

if (CanStackNestedFunc(funcInfo, true))

{

#if DBG

byteCodeFunction->SetCanDoStackNestedFunc();

#endif

if (funcInfo->root->sxFnc.astSize <= PnFnc::MaxStackClosureAST)

{

byteCodeFunction->SetStackNestedFunc(true);

}

}

}

InitScopeSlotArray(funcInfo);

FinalizeRegisters(funcInfo, byteCodeFunction);

DebugOnly(Js::RegSlot firstTmpReg = funcInfo->varRegsCount);

// Reserve temp registers for the inner scopes. We prefer temps because the JIT will then renumber them

// and see different lifetimes. (Note that debug mode requires permanent registers. See FinalizeRegisters.)

uint innerScopeCount = funcInfo->InnerScopeCount();

if (!this->IsInDebugMode())

{

byteCodeFunction->SetInnerScopeCount(innerScopeCount);

if (innerScopeCount)

{

funcInfo->SetFirstInnerScopeReg(funcInfo->AcquireTmpRegister());

for (uint i = 1; i < innerScopeCount; i++)

{

funcInfo->AcquireTmpRegister();

}

}

}

funcInfo->inlineCacheMap = Anew(alloc, FuncInfo::InlineCacheMap,

alloc,

funcInfo->RegCount() // Pass the actual register count. // TODO: Check if we can reduce this count

);

funcInfo->rootObjectLoadInlineCacheMap = Anew(alloc, FuncInfo::RootObjectInlineCacheIdMap,

alloc,

10);

funcInfo->rootObjectLoadMethodInlineCacheMap = Anew(alloc, FuncInfo::RootObjectInlineCacheIdMap,

alloc,

10);

funcInfo->rootObjectStoreInlineCacheMap = Anew(alloc, FuncInfo::RootObjectInlineCacheIdMap,

alloc,

10);

funcInfo->referencedPropertyIdToMapIndex = Anew(alloc, FuncInfo::RootObjectInlineCacheIdMap,

alloc,

10);

byteCodeFunction->AllocateLiteralRegexArray();

m\_callSiteId = 0;

m\_writer.Begin(this, byteCodeFunction, alloc, this->DoJitLoopBodies(funcInfo), funcInfo->hasLoop);

this->PushFuncInfo(L"EmitOneFunction", funcInfo);

this->inPrologue = true;

// Class constructors do not have a [[call]] slot but we don't implement a generic way to express this.

// What we do is emit a check for the new flag here. If we don't have CallFlags\_New set, the opcode will throw.

// We need to do this before emitting 'this' since the base class constructor will try to construct a new object.

if (funcInfo->IsClassConstructor())

{

m\_writer.Empty(Js::OpCode::ChkNewCallFlag);

}

// For now, emit all constant loads at top of function (should instead put in closest dominator of uses).

LoadAllConstants(funcInfo);

HomeArguments(funcInfo);

if (funcInfo->root->sxFnc.pnodeRest != nullptr)

{

byteCodeFunction->SetHasRestParameter();

}

if (funcInfo->thisScopeSlot != Js::Constants::NoRegister && !(funcInfo->IsLambda() || (funcInfo->IsGlobalFunction() && this->flags & fscrEval)))

{

EmitInitCapturedThis(funcInfo, funcInfo->bodyScope);

}

// Any function with a super reference or an eval call inside a class method needs to load super,

if ((funcInfo->HasSuperReference() || (funcInfo->GetCallsEval() && funcInfo->root->sxFnc.IsClassMember()))

// unless we are already inside the 'global' scope inside an eval (in which case 'ScopedLdSuper' is emitted at every 'super' reference).

&& !((GetFlags() & fscrEval) && funcInfo->IsGlobalFunction()))

{

if (funcInfo->IsLambda())

{

Scope \*scope;

Js::PropertyId envIndex = -1;

GetEnclosingNonLambdaScope(funcInfo, scope, envIndex);

FuncInfo\* parent = scope->GetFunc();

if (!parent->IsGlobalFunction())

{

// lambda in non-global scope (eval and non-eval)

EmitInternalScopedSlotLoad(funcInfo, scope, envIndex, parent->superScopeSlot, funcInfo->superRegister);

if (funcInfo->superCtorRegister != Js::Constants::NoRegister)

{

EmitInternalScopedSlotLoad(funcInfo, scope, envIndex, parent->superCtorScopeSlot, funcInfo->superCtorRegister);

}

}

else if (!(GetFlags() & fscrEval))

{

// lambda in non-eval global scope

m\_writer.Reg1(Js::OpCode::LdUndef, funcInfo->superRegister);

}

// lambda in eval global scope: ScopedLdSuper will handle error throwing

}

else

{

m\_writer.Reg1(Js::OpCode::LdSuper, funcInfo->superRegister);

if (funcInfo->superCtorRegister != Js::Constants::NoRegister) // super() is allowed only in derived class constructors

{

m\_writer.Reg1(Js::OpCode::LdSuperCtor, funcInfo->superCtorRegister);

}

if (!funcInfo->IsGlobalFunction())

{

if (funcInfo->bodyScope->GetIsObject() && funcInfo->bodyScope->GetLocation() != Js::Constants::NoRegister)

{

// Stash the super reference in case something inside the eval or lambda references it.

uint cacheId = funcInfo->FindOrAddInlineCacheId(funcInfo->bodyScope->GetLocation(), Js::PropertyIds::\_superReferenceSymbol, false, true);

m\_writer.ElementP(Js::OpCode::InitLocalFld, funcInfo->superRegister, cacheId);

if (funcInfo->superCtorRegister != Js::Constants::NoRegister)

{

cacheId = funcInfo->FindOrAddInlineCacheId(funcInfo->bodyScope->GetLocation(), Js::PropertyIds::\_superCtorReferenceSymbol, false, true);

m\_writer.ElementP(Js::OpCode::InitLocalFld, funcInfo->superCtorRegister, cacheId);

}

}

else if (funcInfo->superScopeSlot == Js::Constants::NoProperty || funcInfo->superCtorScopeSlot == Js::Constants::NoProperty)

{

// While the diag locals walker will pick up super from scoped slots or an activation object,

// it will not pick it up when it is only in a register.

byteCodeFunction->InsertSymbolToRegSlotList(funcInfo->superRegister, Js::PropertyIds::\_superReferenceSymbol, funcInfo->varRegsCount);

if (funcInfo->superCtorRegister != Js::Constants::NoRegister)

{

byteCodeFunction->InsertSymbolToRegSlotList(funcInfo->superCtorRegister, Js::PropertyIds::\_superCtorReferenceSymbol, funcInfo->varRegsCount);

}

}

}

}

}

if (funcInfo->newTargetScopeSlot != Js::Constants::NoRegister && !funcInfo->IsGlobalFunction())

{

EmitInitCapturedNewTarget(funcInfo, funcInfo->bodyScope);

}

// We don't want to load super if we are already in an eval. ScopedLdSuper will take care of loading super in that case.

if (!(GetFlags() & fscrEval) && !funcInfo->bodyScope->GetIsObject())

{

if (funcInfo->superScopeSlot != Js::Constants::NoRegister)

{

this->EmitInternalScopedSlotStore(funcInfo, funcInfo->superScopeSlot, funcInfo->superRegister);

}

if (funcInfo->superCtorScopeSlot != Js::Constants::NoRegister)

{

this->EmitInternalScopedSlotStore(funcInfo, funcInfo->superCtorScopeSlot, funcInfo->superCtorRegister);

}

}

if (byteCodeFunction->DoStackNestedFunc())

{

uint nestedCount = byteCodeFunction->GetNestedCount();

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

{

Js::FunctionProxy \* nested = byteCodeFunction->GetNestedFunc(i);

if (nested->IsFunctionBody())

{

nested->GetFunctionBody()->SetStackNestedFuncParent(byteCodeFunction);

}

}

}

if (scriptContext->GetConfig()->IsLetAndConstEnabled() && funcInfo->IsGlobalFunction())

{

EnsureNoRedeclarations(pnode->sxFnc.pnodeScopes, funcInfo);

}

// Emit all scope-wide function definitions before emitting function bodies

// so that calls may reference functions they precede lexically.

// Note, global eval scope is a fake local scope and is handled as if it were

// a lexical block instead of a true global scope, so do not define the functions

// here. They will be defined during BeginEmitBlock.

if (!(funcInfo->IsGlobalFunction() && this->IsEvalWithBlockScopingNoParentScopeInfo()))

{

DefineFunctions(funcInfo);

}

DefineUserVars(funcInfo);

::BeginEmitBlock(pnode->sxFnc.pnodeScopes, this, funcInfo);

if (pnode->sxFnc.pnodeBodyScope != nullptr)

{

::BeginEmitBlock(pnode->sxFnc.pnodeBodyScope, this, funcInfo);

}

DefineLabels(funcInfo);

if (!pnode->sxFnc.IsSimpleParameterList())

{

EmitDefaultArgs(funcInfo, pnode);

}

else if (funcInfo->GetHasArguments() && !NeedScopeObjectForArguments(funcInfo, pnode))

{

// If we didn't create a scope object and didn't have default args, we still need to transfer the formals to their slots.

MapFormalsWithoutRest(pnode, [&](ParseNode \*pnodeArg) { EmitPropStore(pnodeArg->sxVar.sym->GetLocation(), pnodeArg->sxVar.sym, pnodeArg->sxVar.pid, funcInfo); });

}

// Rest needs to trigger use before declaration until all default args have been processed.

if (pnode->sxFnc.pnodeRest != nullptr)

{

pnode->sxFnc.pnodeRest->sxVar.sym->SetNeedDeclaration(false);

}

this->inPrologue = false;

if (funcInfo->IsGlobalFunction())

{

EmitGlobalBody(funcInfo);

}

else

{

EmitFunctionBody(funcInfo);

}

if (pnode->sxFnc.pnodeBodyScope != nullptr)

{

::EndEmitBlock(pnode->sxFnc.pnodeBodyScope, this, funcInfo);

}

::EndEmitBlock(pnode->sxFnc.pnodeScopes, this, funcInfo);

if (!this->IsInDebugMode())

{

// Release the temp registers that we reserved for inner scopes above.

if (innerScopeCount)

{

Js::RegSlot tmpReg = funcInfo->FirstInnerScopeReg() + innerScopeCount - 1;

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

{

funcInfo->ReleaseTmpRegister(tmpReg);

tmpReg--;

}

}

}

Assert(funcInfo->firstTmpReg == firstTmpReg);

Assert(funcInfo->curTmpReg == firstTmpReg);

Assert(byteCodeFunction->GetFirstTmpReg() == firstTmpReg + byteCodeFunction->GetConstantCount());

byteCodeFunction->SetVarCount(funcInfo->varRegsCount);

byteCodeFunction->SetOutParamDepth(funcInfo->outArgsMaxDepth);

// Do a uint32 add just to verify that we haven't overflowed the reg slot type.

UInt32Math::Add(funcInfo->varRegsCount, funcInfo->constRegsCount);

#if DBG\_DUMP

if (PHASE\_STATS1(Js::ByteCodePhase))

{

Output::Print(L" BCode: %-10d, Aux: %-10d, AuxC: %-10d Total: %-10d, %s\n",

m\_writer.ByteCodeDataSize(),

m\_writer.AuxiliaryDataSize(),

m\_writer.AuxiliaryContextDataSize(),

m\_writer.ByteCodeDataSize() + m\_writer.AuxiliaryDataSize() + m\_writer.AuxiliaryContextDataSize(),

funcInfo->name);

this->scriptContext->byteCodeDataSize += m\_writer.ByteCodeDataSize();

this->scriptContext->byteCodeAuxiliaryDataSize += m\_writer.AuxiliaryDataSize();

this->scriptContext->byteCodeAuxiliaryContextDataSize += m\_writer.AuxiliaryContextDataSize();

}

#endif

this->MapCacheIdsToPropertyIds(funcInfo);

this->MapReferencedPropertyIds(funcInfo);

Assert(this->TopFuncInfo() == funcInfo);

PopFuncInfo(L"EmitOneFunction");

m\_writer.SetCallSiteCount(m\_callSiteId);

#ifdef LOG\_BYTECODE\_AST\_RATIO

m\_writer.End(funcInfo->root->sxFnc.astSize, this->maxAstSize);

#else

m\_writer.End();

#endif

}

catch (...)

{

// Failed to generate byte-code for this function body (likely OOM or stack overflow). Notify the function body so that

// it can revert intermediate state changes that may have taken place during byte code generation before the failure.

byteCodeFunction->ResetByteCodeGenState();

m\_writer.Reset();

throw;

}

#ifdef PERF\_HINT

if (PHASE\_TRACE1(Js::PerfHintPhase) && !byteCodeFunction->GetIsGlobalFunc())

{

if (byteCodeFunction->GetHasTry())

{

WritePerfHint(PerfHints::HasTryBlock\_Verbose, byteCodeFunction);

}

if (funcInfo->GetCallsEval())

{

WritePerfHint(PerfHints::CallsEval\_Verbose, byteCodeFunction);

}

else if (funcInfo->GetChildCallsEval())

{

WritePerfHint(PerfHints::ChildCallsEval, byteCodeFunction);

}

}

#endif

byteCodeFunction->SetInitialDefaultEntryPoint();

byteCodeFunction->SetIsByteCodeDebugMode(this->IsInDebugMode());

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (byteCodeFunction->IsByteCodeDebugMode() != scriptContext->IsInDebugMode()) // debug mode mismatch

{

if (m\_utf8SourceInfo->GetIsLibraryCode())

{

Assert(!byteCodeFunction->IsByteCodeDebugMode()); // Library script byteCode is never in debug mode

}

else

{

Js::Throw::FatalInternalError();

}

}

#endif

#if DBG\_DUMP

if (PHASE\_DUMP(Js::ByteCodePhase, funcInfo->byteCodeFunction) && Js::Configuration::Global.flags.Verbose)

{

pnode->Dump();

}

if (this->Trace() || PHASE\_DUMP(Js::ByteCodePhase, funcInfo->byteCodeFunction))

{

Js::ByteCodeDumper::Dump(byteCodeFunction);

}

if (PHASE\_DUMP(Js::DebuggerScopePhase, funcInfo->byteCodeFunction))

{

byteCodeFunction->DumpScopes();

}

#endif

#if ENABLE\_NATIVE\_CODEGEN

if ((!PHASE\_OFF(Js::BackEndPhase, funcInfo->byteCodeFunction))

&& !this->forceNoNative

&& !this->scriptContext->GetConfig()->IsNoNative())

{

GenerateFunction(this->scriptContext->GetNativeCodeGenerator(), byteCodeFunction);

}

#endif

}

void ByteCodeGenerator::MapCacheIdsToPropertyIds(FuncInfo \*funcInfo)

{

Js::FunctionBody \*functionBody = funcInfo->GetParsedFunctionBody();

uint rootObjectLoadInlineCacheStart = funcInfo->GetInlineCacheCount();

uint rootObjectLoadMethodInlineCacheStart = rootObjectLoadInlineCacheStart + funcInfo->GetRootObjectLoadInlineCacheCount();

uint rootObjectStoreInlineCacheStart = rootObjectLoadMethodInlineCacheStart + funcInfo->GetRootObjectLoadMethodInlineCacheCount();

uint totalFieldAccessInlineCacheCount = rootObjectStoreInlineCacheStart + funcInfo->GetRootObjectStoreInlineCacheCount();

functionBody->CreateCacheIdToPropertyIdMap(rootObjectLoadInlineCacheStart, rootObjectLoadMethodInlineCacheStart,

rootObjectStoreInlineCacheStart, totalFieldAccessInlineCacheCount, funcInfo->GetIsInstInlineCacheCount());

if (totalFieldAccessInlineCacheCount == 0)

{

return;

}

funcInfo->inlineCacheMap->Map([functionBody](Js::RegSlot regSlot, FuncInfo::InlineCacheIdMap \*inlineCacheIdMap)

{

inlineCacheIdMap->Map([functionBody](Js::PropertyId propertyId, FuncInfo::InlineCacheList\* inlineCacheList)

{

if (inlineCacheList)

{

inlineCacheList->Iterate([functionBody, propertyId](InlineCacheUnit cacheUnit)

{

CompileAssert(offsetof(InlineCacheUnit, cacheId) == offsetof(InlineCacheUnit, loadCacheId));

if (cacheUnit.loadCacheId != -1)

{

functionBody->SetPropertyIdForCacheId(cacheUnit.loadCacheId, propertyId);

}

if (cacheUnit.loadMethodCacheId != -1)

{

functionBody->SetPropertyIdForCacheId(cacheUnit.loadMethodCacheId, propertyId);

}

if (cacheUnit.storeCacheId != -1)

{

functionBody->SetPropertyIdForCacheId(cacheUnit.storeCacheId, propertyId);

}

});

}

});

});

funcInfo->rootObjectLoadInlineCacheMap->Map([functionBody, rootObjectLoadInlineCacheStart](Js::PropertyId propertyId, uint cacheId)

{

functionBody->SetPropertyIdForCacheId(cacheId + rootObjectLoadInlineCacheStart, propertyId);

});

funcInfo->rootObjectLoadMethodInlineCacheMap->Map([functionBody, rootObjectLoadMethodInlineCacheStart](Js::PropertyId propertyId, uint cacheId)

{

functionBody->SetPropertyIdForCacheId(cacheId + rootObjectLoadMethodInlineCacheStart, propertyId);

});

funcInfo->rootObjectStoreInlineCacheMap->Map([functionBody, rootObjectStoreInlineCacheStart](Js::PropertyId propertyId, uint cacheId)

{

functionBody->SetPropertyIdForCacheId(cacheId + rootObjectStoreInlineCacheStart, propertyId);

});

SListBase<uint>::Iterator valueOfIter(&funcInfo->valueOfStoreCacheIds);

while (valueOfIter.Next())

{

functionBody->SetPropertyIdForCacheId(valueOfIter.Data(), Js::PropertyIds::valueOf);

}

SListBase<uint>::Iterator toStringIter(&funcInfo->toStringStoreCacheIds);

while (toStringIter.Next())

{

functionBody->SetPropertyIdForCacheId(toStringIter.Data(), Js::PropertyIds::toString);

}

#if DBG

functionBody->VerifyCacheIdToPropertyIdMap();

#endif

}

void ByteCodeGenerator::MapReferencedPropertyIds(FuncInfo \* funcInfo)

{

Js::FunctionBody \*functionBody = funcInfo->GetParsedFunctionBody();

uint referencedPropertyIdCount = funcInfo->GetReferencedPropertyIdCount();

functionBody->CreateReferencedPropertyIdMap(referencedPropertyIdCount);

funcInfo->referencedPropertyIdToMapIndex->Map([functionBody](Js::PropertyId propertyId, uint mapIndex)

{

functionBody->SetReferencedPropertyIdWithMapIndex(mapIndex, propertyId);

});

#if DBG

functionBody->VerifyReferencedPropertyIdMap();

#endif

}

void ByteCodeGenerator::EmitScopeList(ParseNode \*pnode)

{

while (pnode)

{

switch (pnode->nop)

{

case knopFncDecl:

#ifndef TEMP\_DISABLE\_ASMJS

if (pnode->sxFnc.GetAsmjsMode())

{

Js::ExclusiveContext context(this, GetScriptContext());

if (Js::AsmJSCompiler::Compile(&context, pnode, pnode->sxFnc.pnodeArgs))

{

pnode = pnode->sxFnc.pnodeNext;

break;

}

else if (CONFIG\_FLAG(AsmJsStopOnError))

{

exit(JSERR\_AsmJsCompileError);

}

else if (!(flags & fscrDeferFncParse))

{

// If deferral is not allowed, throw and reparse everything with asm.js disabled.

throw Js::AsmJsParseException();

}

}

#endif

// FALLTHROUGH

case knopProg:

if (pnode->sxFnc.funcInfo)

{

this->StartEmitFunction(pnode);

this->EmitScopeList(pnode->sxFnc.pnodeScopes);

this->EmitOneFunction(pnode);

this->EndEmitFunction(pnode);

}

pnode = pnode->sxFnc.pnodeNext;

break;

case knopBlock:

this->StartEmitBlock(pnode);

this->EmitScopeList(pnode->sxBlock.pnodeScopes);

this->EndEmitBlock(pnode);

pnode = pnode->sxBlock.pnodeNext;

break;

case knopCatch:

this->StartEmitCatch(pnode);

this->EmitScopeList(pnode->sxCatch.pnodeScopes);

this->EndEmitCatch(pnode);

pnode = pnode->sxCatch.pnodeNext;

break;

case knopWith:

this->StartEmitWith(pnode);

this->EmitScopeList(pnode->sxWith.pnodeScopes);

this->EndEmitWith(pnode);

pnode = pnode->sxWith.pnodeNext;

break;

default:

AssertMsg(false, "Unexpected opcode in tree of scopes");

break;

}

}

}

void EnsureFncDeclScopeSlot(ParseNode \*pnodeFnc, FuncInfo \*funcInfo)

{

if (pnodeFnc->sxFnc.pnodeName)

{

Assert(pnodeFnc->sxFnc.pnodeName->nop == knopVarDecl);

Symbol \*sym = pnodeFnc->sxFnc.pnodeName->sxVar.sym;

if (sym)

{

sym->EnsureScopeSlot(funcInfo);

}

}

}

// Similar to EnsureFncScopeSlot visitor function, but verifies that a slot is needed before assigning it.

void CheckFncDeclScopeSlot(ParseNode \*pnodeFnc, FuncInfo \*funcInfo)

{

if (pnodeFnc->sxFnc.pnodeName && pnodeFnc->sxFnc.pnodeName->nop == knopVarDecl)

{

Assert(pnodeFnc->sxFnc.pnodeName->nop == knopVarDecl);

Symbol \*sym = pnodeFnc->sxFnc.pnodeName->sxVar.sym;

if (sym && sym->NeedsSlotAlloc(funcInfo))

{

sym->EnsureScopeSlot(funcInfo);

}

}

}

void ByteCodeGenerator::StartEmitFunction(ParseNode \*pnodeFnc)

{

Assert(pnodeFnc->nop == knopFncDecl || pnodeFnc->nop == knopProg);

FuncInfo \*funcInfo = pnodeFnc->sxFnc.funcInfo;

if (funcInfo->byteCodeFunction->IsFunctionParsed() &&

!(flags & (fscrEval | fscrImplicitThis | fscrImplicitParents)))

{

// Only set the environment depth if it's truly known (i.e., not in eval or event handler).

funcInfo->GetParsedFunctionBody()->SetEnvDepth(this->envDepth);

}

if (funcInfo->GetCallsEval())

{

funcInfo->byteCodeFunction->SetDontInline(true);

}

Scope \* const funcExprScope = funcInfo->funcExprScope;

if (funcExprScope)

{

if (funcInfo->GetCallsEval())

{

Assert(funcExprScope->GetIsObject());

}

if (funcExprScope->GetIsObject())

{

funcExprScope->SetCapturesAll(true);

funcExprScope->SetMustInstantiate(true);

PushScope(funcExprScope);

}

else

{

Symbol \*sym = funcInfo->root->sxFnc.GetFuncSymbol();

funcInfo->bodyScope->AddSymbol(sym);

}

}

Scope \* const bodyScope = funcInfo->bodyScope;

if (pnodeFnc->nop != knopProg)

{

if (!bodyScope->GetIsObject() && NeedObjectAsFunctionScope(funcInfo, pnodeFnc))

{

Assert(bodyScope->GetIsObject());

}

if (bodyScope->GetIsObject())

{

bodyScope->SetLocation(funcInfo->frameObjRegister);

}

else

{

bodyScope->SetLocation(funcInfo->frameSlotsRegister);

}

bodyScope->SetMustInstantiate(funcInfo->frameObjRegister != Js::Constants::NoRegister || funcInfo->frameSlotsRegister != Js::Constants::NoRegister);

if (bodyScope->GetIsObject())

{

// Win8 908700: Disable under F12 debugger because there are too many cached scopes holding onto locals.

funcInfo->SetHasCachedScope(

!PHASE\_OFF(Js::CachedScopePhase, funcInfo->byteCodeFunction) &&

!funcInfo->Escapes() &&

funcInfo->frameObjRegister != Js::Constants::NoRegister &&

!ApplyEnclosesArgs(pnodeFnc, this) &&

(PHASE\_FORCE(Js::CachedScopePhase, funcInfo->byteCodeFunction) || !IsInDebugMode()));

if (funcInfo->GetHasCachedScope())

{

Assert(funcInfo->funcObjRegister == Js::Constants::NoRegister);

Symbol \*funcSym = funcInfo->root->sxFnc.GetFuncSymbol();

if (funcSym && funcSym->GetFuncExpr())

{

if (funcSym->GetLocation() == Js::Constants::NoRegister)

{

funcInfo->funcObjRegister = funcInfo->NextVarRegister();

}

else

{

funcInfo->funcObjRegister = funcSym->GetLocation();

}

}

else

{

funcInfo->funcObjRegister = funcInfo->NextVarRegister();

}

Assert(funcInfo->funcObjRegister != Js::Constants::NoRegister);

}

ParseNode \*pnode;

Symbol \*sym;

PushScope(bodyScope);

// Turns on capturesAll temporarily if func has deferred child, so that the following EnsureScopeSlot

// will allocate scope slots no matter if symbol hasNonLocalRefence or not.

Js::ScopeInfo::AutoCapturesAllScope autoCapturesAllScope(bodyScope, funcInfo->HasDeferredChild());

if (funcInfo->GetHasArguments())

{

// Process function's formal parameters

MapFormals(pnodeFnc, [&](ParseNode \*pnode)

{

if (pnode->IsVarLetOrConst())

{

pnode->sxVar.sym->EnsureScopeSlot(funcInfo);

}

});

MapFormalsFromPattern(pnodeFnc, [&](ParseNode \*pnode) { pnode->sxVar.sym->EnsureScopeSlot(funcInfo); });

// Only allocate scope slot for "arguments" when really necessary. "hasDeferredChild"

// doesn't require scope slot for "arguments" because inner functions can't access

// outer function's arguments directly.

sym = funcInfo->GetArgumentsSymbol();

Assert(sym);

if (sym->GetHasNonLocalReference() || autoCapturesAllScope.OldCapturesAll())

{

sym->EnsureScopeSlot(funcInfo);

}

}

sym = funcInfo->root->sxFnc.GetFuncSymbol();

if (sym && sym->NeedsSlotAlloc(funcInfo))

{

if (funcInfo->funcExprScope && funcInfo->funcExprScope->GetIsObject())

{

sym->SetScopeSlot(0);

}

else if (funcInfo->GetFuncExprNameReference())

{

sym->EnsureScopeSlot(funcInfo);

}

}

if (!funcInfo->GetHasArguments())

{

Symbol \*formal;

Js::ArgSlot pos = 1;

auto moveArgToReg = [&](ParseNode \*pnode)

{

if (pnode->IsVarLetOrConst())

{

formal = pnode->sxVar.sym;

// Get the param from its argument position into its assigned register.

// The position should match the location; otherwise, it has been shadowed by parameter with the same name.

if (formal->GetLocation() + 1 == pos)

{

pnode->sxVar.sym->EnsureScopeSlot(funcInfo);

}

}

pos++;

};

MapFormals(pnodeFnc, moveArgToReg);

MapFormalsFromPattern(pnodeFnc, [&](ParseNode \*pnode) { pnode->sxVar.sym->EnsureScopeSlot(funcInfo); });

}

if (funcInfo->isThisLexicallyCaptured)

{

funcInfo->EnsureThisScopeSlot();

}

if (((!funcInfo->IsLambda() && funcInfo->GetCallsEval())

|| funcInfo->isSuperLexicallyCaptured))

{

if (funcInfo->superRegister != Js::Constants::NoRegister)

{

funcInfo->EnsureSuperScopeSlot();

}

if (funcInfo->superCtorRegister != Js::Constants::NoRegister)

{

funcInfo->EnsureSuperCtorScopeSlot();

}

}

if (funcInfo->isNewTargetLexicallyCaptured)

{

funcInfo->EnsureNewTargetScopeSlot();

}

auto ensureFncDeclScopeSlots = [&](ParseNode \*pnodeScope)

{

for (pnode = pnodeScope; pnode;)

{

switch (pnode->nop)

{

case knopFncDecl:

if (pnode->sxFnc.IsDeclaration())

{

EnsureFncDeclScopeSlot(pnode, funcInfo);

}

pnode = pnode->sxFnc.pnodeNext;

break;

case knopBlock:

pnode = pnode->sxBlock.pnodeNext;

break;

case knopCatch:

pnode = pnode->sxCatch.pnodeNext;

break;

case knopWith:

pnode = pnode->sxWith.pnodeNext;

break;

}

}

};

pnodeFnc->sxFnc.MapContainerScopes(ensureFncDeclScopeSlots);

for (pnode = pnodeFnc->sxFnc.pnodeVars; pnode; pnode = pnode->sxVar.pnodeNext)

{

sym = pnode->sxVar.sym;

if (!(pnode->sxVar.isBlockScopeFncDeclVar && sym->GetIsBlockVar()))

{

if (sym->GetIsCatch() || (pnode->nop == knopVarDecl && sym->GetIsBlockVar()))

{

sym = funcInfo->bodyScope->FindLocalSymbol(sym->GetName());

}

if (sym->GetSymbolType() == STVariable && !sym->GetIsArguments())

{

sym->EnsureScopeSlot(funcInfo);

}

}

}

if (pnodeFnc->sxFnc.pnodeBody)

{

Assert(pnodeFnc->sxFnc.pnodeScopes->nop == knopBlock);

this->EnsureLetConstScopeSlots(pnodeFnc->sxFnc.pnodeBodyScope, funcInfo);

}

}

else

{

ParseNode \*pnode;

Symbol \*sym;

// Don't rely on the Emit() pass to assign scope slots where needed, because peeps/shortcuts

// may cause some expressions not to be emitted. Assign the slots we need before we start

// emitting the prolog.

// TODO: Investigate moving detection of non-local references to Emit() so we don't assign

// slots to symbols that are never referenced in emitted code.

if (funcInfo->isThisLexicallyCaptured)

{

funcInfo->EnsureThisScopeSlot();

}

if (funcInfo->isSuperLexicallyCaptured)

{

funcInfo->EnsureSuperScopeSlot();

}

if (funcInfo->isSuperCtorLexicallyCaptured)

{

funcInfo->EnsureSuperCtorScopeSlot();

}

if (funcInfo->isNewTargetLexicallyCaptured)

{

funcInfo->EnsureNewTargetScopeSlot();

}

pnodeFnc->sxFnc.MapContainerScopes([&](ParseNode \*pnodeScope) { this->EnsureFncScopeSlots(pnodeScope, funcInfo); });

PushScope(bodyScope);

for (pnode = pnodeFnc->sxFnc.pnodeVars; pnode; pnode = pnode->sxVar.pnodeNext)

{

sym = pnode->sxVar.sym;

if (!(pnode->sxVar.isBlockScopeFncDeclVar && sym->GetIsBlockVar()))

{

if (sym->GetIsCatch() || (pnode->nop == knopVarDecl && sym->GetIsBlockVar()))

{

sym = funcInfo->bodyScope->FindLocalSymbol(sym->GetName());

}

if (sym->GetSymbolType() == STVariable && sym->NeedsSlotAlloc(funcInfo) && !sym->GetIsArguments())

{

sym->EnsureScopeSlot(funcInfo);

}

}

}

auto ensureScopeSlot = [&](ParseNode \*pnode)

{

if (pnode->IsVarLetOrConst())

{

sym = pnode->sxVar.sym;

if (sym->GetSymbolType() == STFormal && sym->NeedsSlotAlloc(funcInfo))

{

sym->EnsureScopeSlot(funcInfo);

}

}

};

// Process function's formal parameters

MapFormals(pnodeFnc, ensureScopeSlot);

MapFormalsFromPattern(pnodeFnc, ensureScopeSlot);

if (pnodeFnc->sxFnc.pnodeBody)

{

this->EnsureLetConstScopeSlots(pnodeFnc->sxFnc.pnodeScopes, funcInfo);

this->EnsureLetConstScopeSlots(pnodeFnc->sxFnc.pnodeBodyScope, funcInfo);

}

}

}

else

{

bool newScopeForEval = (funcInfo->byteCodeFunction->GetIsStrictMode() && (this->GetFlags() & fscrEval));

if (newScopeForEval)

{

Assert(bodyScope->GetIsObject());

}

PushScope(bodyScope);

}

// Persist outer func scope info if nested func is deferred

if (CONFIG\_FLAG(DeferNested))

{

FuncInfo\* parentFunc = TopFuncInfo();

Js::ScopeInfo::SaveScopeInfoForDeferParse(this, parentFunc, funcInfo);

PushFuncInfo(L"StartEmitFunction", funcInfo);

}

}

void ByteCodeGenerator::EnsureLetConstScopeSlots(ParseNode \*pnodeBlock, FuncInfo \*funcInfo)

{

bool hasNonLocalReference = pnodeBlock->sxBlock.GetCallsEval() || pnodeBlock->sxBlock.GetChildCallsEval();

auto ensureLetConstSlots = ([this, pnodeBlock, funcInfo, hasNonLocalReference](ParseNode \*pnode)

{

Symbol \*sym = pnode->sxVar.sym;

sym->EnsureScopeSlot(funcInfo);

if (hasNonLocalReference)

{

sym->SetHasNonLocalReference(true, this);

}

});

IterateBlockScopedVariables(pnodeBlock, ensureLetConstSlots);

}

void ByteCodeGenerator::EnsureFncScopeSlots(ParseNode \*pnode, FuncInfo \*funcInfo)

{

while (pnode)

{

switch (pnode->nop)

{

case knopFncDecl:

if (pnode->sxFnc.IsDeclaration())

{

CheckFncDeclScopeSlot(pnode, funcInfo);

}

pnode = pnode->sxFnc.pnodeNext;

break;

case knopBlock:

pnode = pnode->sxBlock.pnodeNext;

break;

case knopCatch:

pnode = pnode->sxCatch.pnodeNext;

break;

case knopWith:

pnode = pnode->sxWith.pnodeNext;

break;

}

}

}

void ByteCodeGenerator::EndEmitFunction(ParseNode \*pnodeFnc)

{

Assert(pnodeFnc->nop == knopFncDecl || pnodeFnc->nop == knopProg);

Assert(pnodeFnc->nop == knopFncDecl && currentScope->GetEnclosingScope() != nullptr || pnodeFnc->nop == knopProg);

PopScope(); // function body

FuncInfo \*funcInfo = pnodeFnc->sxFnc.funcInfo;

Scope \*scope = funcInfo->funcExprScope;

if (scope && scope->GetMustInstantiate())

{

Assert(currentScope == scope);

PopScope();

}

if (CONFIG\_FLAG(DeferNested))

{

Assert(funcInfo == this->TopFuncInfo());

PopFuncInfo(L"EndEmitFunction");

}

}

void ByteCodeGenerator::StartEmitCatch(ParseNode \*pnodeCatch)

{

Assert(pnodeCatch->nop == knopCatch);

Scope \*scope = pnodeCatch->sxCatch.scope;

FuncInfo \*funcInfo = scope->GetFunc();

// Catch scope is a dynamic object if it can be passed to a scoped lookup helper (i.e., eval is present or we're in an event handler).

if (funcInfo->GetCallsEval() || funcInfo->GetChildCallsEval() || (this->flags & (fscrEval | fscrImplicitThis | fscrImplicitParents)))

{

scope->SetIsObject();

}

if (pnodeCatch->sxCatch.pnodeParam->nop == knopParamPattern)

{

scope->SetCapturesAll(funcInfo->GetCallsEval() || funcInfo->GetChildCallsEval());

scope->SetMustInstantiate(scope->GetMustInstantiate() || scope->GetCapturesAll() || funcInfo->IsGlobalFunction());

Parser::MapBindIdentifier(pnodeCatch->sxCatch.pnodeParam->sxParamPattern.pnode1, [&](ParseNodePtr item)

{

Symbol \*sym = item->sxVar.sym;

if (funcInfo->IsGlobalFunction())

{

sym->SetIsGlobalCatch(true);

}

Assert(sym->GetScopeSlot() == Js::Constants::NoProperty);

if (sym->NeedsSlotAlloc(funcInfo))

{

sym->EnsureScopeSlot(funcInfo);

}

});

// In the case of pattern we will always going to push the scope.

PushScope(scope);

}

else

{

Symbol \*sym = pnodeCatch->sxCatch.pnodeParam->sxPid.sym;

// Catch object is stored in the catch scope if there may be an ambiguous lookup or a var declaration that hides it.

scope->SetCapturesAll(funcInfo->GetCallsEval() || funcInfo->GetChildCallsEval() || sym->GetHasNonLocalReference());

scope->SetMustInstantiate(scope->GetCapturesAll() || funcInfo->IsGlobalFunction() || currentScope != funcInfo->GetBodyScope());

if (funcInfo->IsGlobalFunction())

{

sym->SetIsGlobalCatch(true);

}

if (scope->GetMustInstantiate())

{

// Since there is only one symbol we are pushing to slot.

// Also in order to make IsInSlot to return true - forcing the sym-has-non-local-reference.

sym->SetHasNonLocalReference(true, this);

sym->EnsureScopeSlot(funcInfo);

PushScope(scope);

}

else

{

// Add it to the parent function's scope and treat it like any other local.

// We can only do this if we don't need to get the symbol from a slot, though, because adding it to the

// parent's scope object on entry to the catch could re-size the slot array.

funcInfo->bodyScope->AddSymbol(sym);

}

}

}

void ByteCodeGenerator::EndEmitCatch(ParseNode \*pnodeCatch)

{

Assert(pnodeCatch->nop == knopCatch);

if (pnodeCatch->sxCatch.scope->GetMustInstantiate() || pnodeCatch->sxCatch.pnodeParam->nop == knopParamPattern)

{

Assert(currentScope == pnodeCatch->sxCatch.scope);

PopScope();

}

}

void ByteCodeGenerator::StartEmitBlock(ParseNode \*pnodeBlock)

{

if (!BlockHasOwnScope(pnodeBlock, this))

{

return;

}

Assert(pnodeBlock->nop == knopBlock);

PushBlock(pnodeBlock);

Scope \*scope = pnodeBlock->sxBlock.scope;

if (pnodeBlock->sxBlock.GetCallsEval() || pnodeBlock->sxBlock.GetChildCallsEval() || (this->flags & (fscrEval | fscrImplicitThis | fscrImplicitParents)))

{

Assert(scope->GetIsObject());

}

// TODO: Consider nested deferred parsing.

if (scope->GetMustInstantiate())

{

FuncInfo \*funcInfo = scope->GetFunc();

if (scope->IsGlobalEvalBlockScope() && funcInfo->isThisLexicallyCaptured)

{

funcInfo->EnsureThisScopeSlot();

}

this->EnsureFncScopeSlots(pnodeBlock->sxBlock.pnodeScopes, funcInfo);

this->EnsureLetConstScopeSlots(pnodeBlock, funcInfo);

PushScope(scope);

}

}

void ByteCodeGenerator::EndEmitBlock(ParseNode \*pnodeBlock)

{

if (!BlockHasOwnScope(pnodeBlock, this))

{

return;

}

Assert(pnodeBlock->nop == knopBlock);

Scope \*scope = pnodeBlock->sxBlock.scope;

if (scope && scope->GetMustInstantiate())

{

Assert(currentScope == pnodeBlock->sxBlock.scope);

PopScope();

}

PopBlock();

}

void ByteCodeGenerator::StartEmitWith(ParseNode \*pnodeWith)

{

Assert(pnodeWith->nop == knopWith);

Scope \*scope = pnodeWith->sxWith.scope;

Assert(scope->GetIsObject());

PushScope(scope);

}

void ByteCodeGenerator::EndEmitWith(ParseNode \*pnodeWith)

{

Assert(pnodeWith->nop == knopWith);

Assert(currentScope == pnodeWith->sxWith.scope);

PopScope();

}

Js::RegSlot ByteCodeGenerator::PrependLocalScopes(Js::RegSlot evalEnv, Js::RegSlot tempLoc, FuncInfo \*funcInfo)

{

Scope \*currScope = this->currentScope;

Scope \*funcScope = funcInfo->GetBodyScope();

if (currScope == funcScope)

{

return evalEnv;

}

bool acquireTempLoc = tempLoc == Js::Constants::NoRegister;

if (acquireTempLoc)

{

tempLoc = funcInfo->AcquireTmpRegister();

}

// The with/catch objects must be prepended to the environment we pass to eval() or to a func declared inside with,

// but the list must first be reversed so that innermost scopes appear first in the list.

while (currScope != funcScope)

{

Scope \*innerScope;

for (innerScope = currScope; innerScope->GetEnclosingScope() != funcScope; innerScope = innerScope->GetEnclosingScope())

;

if (innerScope->GetMustInstantiate())

{

if (!innerScope->HasInnerScopeIndex())

{

if (evalEnv == funcInfo->GetEnvRegister() || evalEnv == funcInfo->frameDisplayRegister)

{

this->m\_writer.Reg2(Js::OpCode::LdInnerFrameDisplayNoParent, tempLoc, innerScope->GetLocation());

}

else

{

this->m\_writer.Reg3(Js::OpCode::LdInnerFrameDisplay, tempLoc, innerScope->GetLocation(), evalEnv);

}

}

else

{

if (evalEnv == funcInfo->GetEnvRegister() || evalEnv == funcInfo->frameDisplayRegister)

{

this->m\_writer.Reg1Unsigned1(Js::OpCode::LdIndexedFrameDisplayNoParent, tempLoc, innerScope->GetInnerScopeIndex());

}

else

{

this->m\_writer.Reg2Int1(Js::OpCode::LdIndexedFrameDisplay, tempLoc, evalEnv, innerScope->GetInnerScopeIndex());

}

}

evalEnv = tempLoc;

}

funcScope = innerScope;

}

if (acquireTempLoc)

{

funcInfo->ReleaseTmpRegister(tempLoc);

}

return evalEnv;

}

void ByteCodeGenerator::EmitLoadInstance(Symbol \*sym, IdentPtr pid, Js::RegSlot \*pThisLocation, Js::RegSlot \*pInstLocation, FuncInfo \*funcInfo)

{

Js::ByteCodeLabel doneLabel = 0;

bool fLabelDefined = false;

Js::RegSlot scopeLocation = Js::Constants::NoRegister;

Js::RegSlot thisLocation = \*pThisLocation;

Js::RegSlot instLocation = \*pInstLocation;

Js::PropertyId envIndex = -1;

Scope \*scope = nullptr;

Scope \*symScope = sym ? sym->GetScope() : this->globalScope;

Assert(symScope);

for (;;)

{

scope = this->FindScopeForSym(symScope, scope, &envIndex, funcInfo);

if (scope == this->globalScope)

{

break;

}

if (scope != symScope)

{

// We're not sure where the function is (eval/with/etc).

// So we're going to need registers to hold the instance where we (dynamically) find

// the function, and possibly to hold the "this" pointer we will pass to it.

// Assign them here so that they can't overlap with the scopeLocation assigned below.

// Otherwise we wind up with temp lifetime confusion in the IRBuilder. (Win8 281689)

if (instLocation == Js::Constants::NoRegister)

{

instLocation = funcInfo->AcquireTmpRegister();

// The "this" pointer will not be the same as the instance, so give it its own register.

thisLocation = funcInfo->AcquireTmpRegister();

}

}

if (envIndex == -1)

{

Assert(funcInfo == scope->GetFunc());

scopeLocation = scope->GetLocation();

}

if (scope == symScope)

{

break;

}

// Found a scope to which the property may have been added.

Assert(scope && scope->GetIsDynamic());

if (!fLabelDefined)

{

fLabelDefined = true;

doneLabel = this->m\_writer.DefineLabel();

}

Js::ByteCodeLabel nextLabel = this->m\_writer.DefineLabel();

Js::PropertyId propertyId = sym ? sym->EnsurePosition(this) : pid->GetPropertyId();

bool unwrapWithObj = scope->GetScopeType() == ScopeType\_With && scriptContext->GetConfig()->IsES6UnscopablesEnabled();

if (envIndex != -1)

{

this->m\_writer.BrEnvProperty(

Js::OpCode::BrOnNoEnvProperty, nextLabel,

funcInfo->FindOrAddReferencedPropertyId(propertyId),

envIndex + Js::FrameDisplay::GetOffsetOfScopes()/sizeof(Js::Var));

Js::RegSlot tmpReg = funcInfo->AcquireTmpRegister();

this->m\_writer.SlotI1(Js::OpCode::LdEnvObj, tmpReg,

envIndex + Js::FrameDisplay::GetOffsetOfScopes()/sizeof(Js::Var));

Js::OpCode op = unwrapWithObj ? Js::OpCode::UnwrapWithObj : Js::OpCode::Ld\_A;

this->m\_writer.Reg2(op, instLocation, tmpReg);

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

{

this->m\_writer.Reg2(op, thisLocation, tmpReg);

}

funcInfo->ReleaseTmpRegister(tmpReg);

}

else if (scopeLocation != Js::Constants::NoRegister && scopeLocation == funcInfo->frameObjRegister)

{

this->m\_writer.BrLocalProperty(Js::OpCode::BrOnNoLocalProperty, nextLabel,

funcInfo->FindOrAddReferencedPropertyId(propertyId));

Assert(!unwrapWithObj);

this->m\_writer.Reg1(Js::OpCode::LdLocalObj, instLocation);

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

{

this->m\_writer.Reg1(Js::OpCode::LdLocalObj, thisLocation);

}

}

else

{

this->m\_writer.BrProperty(Js::OpCode::BrOnNoProperty, nextLabel, scopeLocation,

funcInfo->FindOrAddReferencedPropertyId(propertyId));

Js::OpCode op = unwrapWithObj ? Js::OpCode::UnwrapWithObj : Js::OpCode::Ld\_A;

this->m\_writer.Reg2(op, instLocation, scopeLocation);

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

{

this->m\_writer.Reg2(op, thisLocation, scopeLocation);

}

}

this->m\_writer.Br(doneLabel);

this->m\_writer.MarkLabel(nextLabel);

}

if (sym == nullptr || sym->GetIsGlobal())

{

if (this->flags & (fscrEval | fscrImplicitThis | fscrImplicitParents))

{

// Load of a symbol with unknown scope from within eval.

// Get it from the closure environment.

if (instLocation == Js::Constants::NoRegister)

{

instLocation = funcInfo->AcquireTmpRegister();

}

// TODO: It should be possible to avoid this double call to ScopedLdInst by having it return both

// results at once. The reason for the uncertainty here is that we don't know whether the callee

// belongs to a "with" object. If it does, we have to pass the "with" object as "this"; in all other

// cases, we pass "undefined". For now, there are apparently no significant performance issues.

Js::PropertyId propertyId = sym ? sym->EnsurePosition(this) : pid->GetPropertyId();

if (thisLocation == Js::Constants::NoRegister)

{

thisLocation = funcInfo->AcquireTmpRegister();

}

this->m\_writer.ScopedProperty2(Js::OpCode::ScopedLdInst, instLocation,

funcInfo->FindOrAddReferencedPropertyId(propertyId), thisLocation);

}

else

{

if (instLocation == Js::Constants::NoRegister)

{

instLocation = ByteCodeGenerator::RootObjectRegister;

}

else

{

this->m\_writer.Reg2(Js::OpCode::Ld\_A, instLocation, ByteCodeGenerator::RootObjectRegister);

}

if (thisLocation == Js::Constants::NoRegister)

{

thisLocation = funcInfo->undefinedConstantRegister;

}

else

{

this->m\_writer.Reg2(Js::OpCode::Ld\_A, thisLocation, funcInfo->undefinedConstantRegister);

}

}

}

else if (instLocation != Js::Constants::NoRegister)

{

if (envIndex != -1)

{

this->m\_writer.SlotI1(Js::OpCode::LdEnvObj, instLocation,

envIndex + Js::FrameDisplay::GetOffsetOfScopes()/sizeof(Js::Var));

}

else if (scope->HasInnerScopeIndex())

{

this->m\_writer.Reg1Unsigned1(Js::OpCode::LdInnerScope, instLocation, scope->GetInnerScopeIndex());

}

else if (symScope != funcInfo->GetBodyScope())

{

this->m\_writer.Reg2(Js::OpCode::Ld\_A, instLocation, scopeLocation);

}

else

{

this->m\_writer.Reg1(Js::OpCode::LdLocalObj, instLocation);

}

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

{

this->m\_writer.Reg2(Js::OpCode::Ld\_A, thisLocation, funcInfo->undefinedConstantRegister);

}

else

{

thisLocation = funcInfo->undefinedConstantRegister;

}

}

\*pThisLocation = thisLocation;

\*pInstLocation = instLocation;

if (fLabelDefined)

{

this->m\_writer.MarkLabel(doneLabel);

}

}

void ByteCodeGenerator::EmitGlobalFncDeclInit(Js::RegSlot rhsLocation, Js::PropertyId propertyId, FuncInfo \* funcInfo)

{

// Note: declared variables and assignments in the global function go to the root object directly.

if (this->flags & fscrEval)

{

// Func decl's always get their init values before any use, so we don't pre-initialize the property to undef.

// That means that we have to use ScopedInitFld so that we initialize the property on the right instance

// even if the instance doesn't have the property yet (i.e., collapse the init-to-undef and the store

// into one operation). See WOOB 1121763 and 1120973.

this->m\_writer.ScopedProperty(Js::OpCode::ScopedInitFunc, rhsLocation,

funcInfo->FindOrAddReferencedPropertyId(propertyId));

}

else

{

this->EmitPatchableRootProperty(Js::OpCode::InitRootFld, rhsLocation, propertyId, false, true, funcInfo);

}

}

void

ByteCodeGenerator::EmitPatchableRootProperty(Js::OpCode opcode,

Js::RegSlot regSlot, Js::PropertyId propertyId, bool isLoadMethod, bool isStore, FuncInfo \* funcInfo)

{

uint cacheId = funcInfo->FindOrAddRootObjectInlineCacheId(propertyId, isLoadMethod, isStore);

this->m\_writer.PatchableRootProperty(opcode, regSlot, cacheId, isLoadMethod, isStore);

}

void ByteCodeGenerator::EmitLocalPropInit(Js::RegSlot rhsLocation, Symbol \*sym, FuncInfo \*funcInfo)

{

Scope \*scope = sym->GetScope();

// Check consistency of sym->IsInSlot.

Assert(sym->NeedsSlotAlloc(funcInfo) || sym->GetScopeSlot() == Js::Constants::NoProperty);

// Arrived at the scope in which the property was defined.

if (sym->NeedsSlotAlloc(funcInfo))

{

// The property is in memory rather than register. We'll have to load it from the slots.

if (scope->GetIsObject())

{

Assert(!this->TopFuncInfo()->GetParsedFunctionBody()->DoStackNestedFunc());

Js::PropertyId propertyId = sym->EnsurePosition(this);

Js::RegSlot objReg;

if (scope->HasInnerScopeIndex())

{

objReg = funcInfo->InnerScopeToRegSlot(scope);

}

else

{

objReg = scope->GetLocation();

}

uint cacheId = funcInfo->FindOrAddInlineCacheId(objReg, propertyId, false, true);

Js::OpCode op = this->GetInitFldOp(scope, objReg, funcInfo, sym->GetIsNonSimpleParameter());

if (objReg != Js::Constants::NoRegister && objReg == funcInfo->frameObjRegister)

{

this->m\_writer.ElementP(op, rhsLocation, cacheId);

}

else if (scope->HasInnerScopeIndex())

{

this->m\_writer.ElementPIndexed(op, rhsLocation, scope->GetInnerScopeIndex(), cacheId);

}

else

{

this->m\_writer.PatchableProperty(op, rhsLocation, scope->GetLocation(), cacheId);

}

}

else

{

// Make sure the property has a slot. This will bump up the size of the slot array if necessary.

Js::PropertyId slot = sym->EnsureScopeSlot(funcInfo);

Js::RegSlot slotReg = scope->GetCanMerge() ? funcInfo->frameSlotsRegister : scope->GetLocation();

// Now store the property to its slot.

Js::OpCode op = this->GetStSlotOp(scope, -1, slotReg, false, funcInfo);

if (slotReg != Js::Constants::NoRegister && slotReg == funcInfo->frameSlotsRegister)

{

this->m\_writer.SlotI1(op, rhsLocation, slot + Js::ScopeSlots::FirstSlotIndex);

}

else

{

this->m\_writer.SlotI2(op, rhsLocation, scope->GetInnerScopeIndex(), slot + Js::ScopeSlots::FirstSlotIndex);

}

}

}

if (sym->GetLocation() != Js::Constants::NoRegister && rhsLocation != sym->GetLocation())

{

this->m\_writer.Reg2(Js::OpCode::Ld\_A, sym->GetLocation(), rhsLocation);

}

}

Js::OpCode

ByteCodeGenerator::GetStSlotOp(Scope \*scope, int envIndex, Js::RegSlot scopeLocation, bool chkBlockVar, FuncInfo \*funcInfo)

{

Js::OpCode op;

if (envIndex != -1)

{

if (scope->GetIsObject())

{

op = Js::OpCode::StEnvObjSlot;

}

else

{

op = Js::OpCode::StEnvSlot;

}

}

else if (scopeLocation != Js::Constants::NoRegister &&

scopeLocation == funcInfo->frameSlotsRegister)

{

op = Js::OpCode::StLocalSlot;

}

else if (scopeLocation != Js::Constants::NoRegister &&

scopeLocation == funcInfo->frameObjRegister)

{

op = Js::OpCode::StLocalObjSlot;

}

else

{

Assert(scope->HasInnerScopeIndex());

if (scope->GetIsObject())

{

op = Js::OpCode::StInnerObjSlot;

}

else

{

op = Js::OpCode::StInnerSlot;

}

}

if (chkBlockVar)

{

op = this->ToChkUndeclOp(op);

}

return op;

}

Js::OpCode

ByteCodeGenerator::GetInitFldOp(Scope \*scope, Js::RegSlot scopeLocation, FuncInfo \*funcInfo, bool letDecl)

{

Js::OpCode op;

if (scopeLocation != Js::Constants::NoRegister &&

scopeLocation == funcInfo->frameObjRegister)

{

op = letDecl ? Js::OpCode::InitLocalLetFld : Js::OpCode::InitLocalFld;

}

else if (scope->HasInnerScopeIndex())

{

op = letDecl ? Js::OpCode::InitInnerLetFld : Js::OpCode::InitInnerFld;

}

else

{

op = letDecl ? Js::OpCode::InitLetFld : Js::OpCode::InitFld;

}

return op;

}

void ByteCodeGenerator::EmitPropStore(Js::RegSlot rhsLocation, Symbol \*sym, IdentPtr pid, FuncInfo \*funcInfo, bool isLetDecl, bool isConstDecl, bool isFncDeclVar)

{

Js::ByteCodeLabel doneLabel = 0;

bool fLabelDefined = false;

Js::PropertyId envIndex = -1;

Scope \*symScope = sym == nullptr || sym->GetIsGlobal() ? this->globalScope : sym->GetScope();

Assert(symScope);

// isFncDeclVar denotes that the symbol being stored to here is the var

// binding of a function declaration and we know we want to store directly

// to it, skipping over any dynamic scopes that may lie in between.

Scope \*scope = isFncDeclVar ? symScope : nullptr;

Js::RegSlot scopeLocation = isFncDeclVar ? scope->GetLocation() : Js::Constants::NoRegister;

bool scopeAcquired = false;

Js::OpCode op;

while (!isFncDeclVar)

{

scope = this->FindScopeForSym(symScope, scope, &envIndex, funcInfo);

if (scope == this->globalScope)

{

break;

}

if (envIndex == -1)

{

Assert(funcInfo == scope->GetFunc());

scopeLocation = scope->GetLocation();

}

if (scope == symScope)

{

break;

}

// Found a scope to which the property may have been added.

Assert(scope && scope->GetIsDynamic());

if (!fLabelDefined)

{

fLabelDefined = true;

doneLabel = this->m\_writer.DefineLabel();

}

Js::ByteCodeLabel nextLabel = this->m\_writer.DefineLabel();

Js::PropertyId propertyId = sym ? sym->EnsurePosition(this) : pid->GetPropertyId();

Js::RegSlot unwrappedScopeLocation = scopeLocation;

bool unwrapWithObj = scope->GetScopeType() == ScopeType\_With && scriptContext->GetConfig()->IsES6UnscopablesEnabled();

if (envIndex != -1)

{

this->m\_writer.BrEnvProperty(

Js::OpCode::BrOnNoEnvProperty,

nextLabel,

funcInfo->FindOrAddReferencedPropertyId(propertyId),

envIndex + Js::FrameDisplay::GetOffsetOfScopes()/sizeof(Js::Var));

Js::RegSlot instLocation = funcInfo->AcquireTmpRegister();

this->m\_writer.SlotI1(

Js::OpCode::LdEnvObj,

instLocation,

envIndex + Js::FrameDisplay::GetOffsetOfScopes()/sizeof(Js::Var));

if (unwrapWithObj)

{

this->m\_writer.Reg2(Js::OpCode::UnwrapWithObj, instLocation, instLocation);

}

this->m\_writer.PatchableProperty(

Js::OpCode::StFld,

rhsLocation,

instLocation,

funcInfo->FindOrAddInlineCacheId(instLocation, propertyId, false, true));

funcInfo->ReleaseTmpRegister(instLocation);

}

else if (scopeLocation != Js::Constants::NoRegister && scopeLocation == funcInfo->frameObjRegister)

{

this->m\_writer.BrLocalProperty(Js::OpCode::BrOnNoLocalProperty, nextLabel,

funcInfo->FindOrAddReferencedPropertyId(propertyId));

Assert(!unwrapWithObj);

this->m\_writer.ElementP(Js::OpCode::StLocalFld, rhsLocation,

funcInfo->FindOrAddInlineCacheId(scopeLocation, propertyId, false, true));

}

else

{

this->m\_writer.BrProperty(Js::OpCode::BrOnNoProperty, nextLabel, scopeLocation,

funcInfo->FindOrAddReferencedPropertyId(propertyId));

if (unwrapWithObj)

{

unwrappedScopeLocation = funcInfo->AcquireTmpRegister();

this->m\_writer.Reg2(Js::OpCode::UnwrapWithObj, unwrappedScopeLocation, scopeLocation);

scopeLocation = unwrappedScopeLocation;

}

uint cacheId = funcInfo->FindOrAddInlineCacheId(scopeLocation, propertyId, false, true);

this->m\_writer.PatchableProperty(Js::OpCode::StFld, rhsLocation, scopeLocation, cacheId);

if (unwrapWithObj)

{

funcInfo->ReleaseTmpRegister(unwrappedScopeLocation);

}

}

this->m\_writer.Br(doneLabel);

this->m\_writer.MarkLabel(nextLabel);

}

// Arrived at the scope in which the property was defined.

if (sym == nullptr || sym->GetIsGlobal())

{

Js::PropertyId propertyId = sym ? sym->EnsurePosition(this) : pid->GetPropertyId();

if (this->flags & fscrEval)

{

if (funcInfo->byteCodeFunction->GetIsStrictMode() && funcInfo->IsGlobalFunction())

{

uint cacheId = funcInfo->FindOrAddInlineCacheId(funcInfo->frameDisplayRegister, propertyId, false, true);

this->m\_writer.ElementP(GetScopedStFldOpCode(funcInfo), rhsLocation, cacheId);

}

else

{

uint cacheId = funcInfo->FindOrAddInlineCacheId(funcInfo->GetEnvRegister(), propertyId, false, true);

bool isConsoleScopeLetConst = this->IsConsoleScopeEval() && (isLetDecl || isConstDecl);

// In "eval", store to a symbol with unknown scope goes through the closure environment.

this->m\_writer.ElementP(GetScopedStFldOpCode(funcInfo, isConsoleScopeLetConst), rhsLocation, cacheId);

}

}

else if (this->flags & (fscrImplicitThis | fscrImplicitParents))

{

uint cacheId = funcInfo->FindOrAddInlineCacheId(funcInfo->GetEnvRegister(), propertyId, false, true);

// In "eval", store to a symbol with unknown scope goes through the closure environment.

this->m\_writer.ElementP(GetScopedStFldOpCode(funcInfo), rhsLocation, cacheId);

}

else

{

this->EmitPatchableRootProperty(GetStFldOpCode(funcInfo, true, isLetDecl, isConstDecl, false), rhsLocation, propertyId, false, true, funcInfo);

}

}

else if (sym->GetFuncExpr())

{

// Store to function expr variable.

// strict mode: we need to throw type error

if (funcInfo->byteCodeFunction->GetIsStrictMode())

{

// Note that in this case the sym's location belongs to the parent function, so we can't use it.

// It doesn't matter which register we use, as long as it's valid for this function.

this->m\_writer.W1(Js::OpCode::RuntimeTypeError, SCODE\_CODE(JSERR\_CantAssignToReadOnly));

}

}

else if (sym->IsInSlot(funcInfo) || envIndex != -1)

{

if (!isConstDecl && sym->GetDecl() && sym->GetDecl()->nop == knopConstDecl)

{

// This is a case where const reassignment can't be proven statically (e.g., eval, with) so

// we have to catch it at runtime.

this->m\_writer.W1(

Js::OpCode::RuntimeReferenceError, SCODE\_CODE(ERRAssignmentToConst));

}

// Make sure the property has a slot. This will bump up the size of the slot array if necessary.

Js::PropertyId slot = sym->EnsureScopeSlot(funcInfo);

bool chkBlockVar = !isLetDecl && !isConstDecl && NeedCheckBlockVar(sym, scope, funcInfo);

// The property is in memory rather than register. We'll have to load it from the slots.

op = this->GetStSlotOp(scope, envIndex, scopeLocation, chkBlockVar, funcInfo);

if (envIndex != -1)

{

this->m\_writer.SlotI2(op, rhsLocation,

envIndex + Js::FrameDisplay::GetOffsetOfScopes()/sizeof(Js::Var),

slot + (sym->GetScope()->GetIsObject()? 0 : Js::ScopeSlots::FirstSlotIndex));

}

else if (scopeLocation != Js::Constants::NoRegister &&

(scopeLocation == funcInfo->frameSlotsRegister || scopeLocation == funcInfo->frameObjRegister))

{

this->m\_writer.SlotI1(op, rhsLocation,

slot + (sym->GetScope()->GetIsObject()? 0 : Js::ScopeSlots::FirstSlotIndex));

}

else

{

Assert(scope->HasInnerScopeIndex());

this->m\_writer.SlotI2(op, rhsLocation, scope->GetInnerScopeIndex(),

slot + (sym->GetScope()->GetIsObject()? 0 : Js::ScopeSlots::FirstSlotIndex));

}

if (this->ShouldTrackDebuggerMetadata() && (isLetDecl || isConstDecl))

{

Js::PropertyId location = scope->GetIsObject() ? sym->GetLocation() : slot;

this->UpdateDebuggerPropertyInitializationOffset(location, sym->GetPosition(), false);

}

}

else if (isConstDecl)

{

this->m\_writer.Reg2(Js::OpCode::InitConst, sym->GetLocation(), rhsLocation);

if (this->ShouldTrackDebuggerMetadata())

{

this->UpdateDebuggerPropertyInitializationOffset(sym->GetLocation(), sym->GetPosition());

}

}

else

{

if (!isConstDecl && sym->GetDecl() && sym->GetDecl()->nop == knopConstDecl)

{

// This is a case where const reassignment can't be proven statically (e.g., eval, with) so

// we have to catch it at runtime.

this->m\_writer.W1(Js::OpCode::RuntimeReferenceError, SCODE\_CODE(ERRAssignmentToConst));

}

if (rhsLocation != sym->GetLocation())

{

this->m\_writer.Reg2(Js::OpCode::Ld\_A, sym->GetLocation(), rhsLocation);

if (this->ShouldTrackDebuggerMetadata() && isLetDecl)

{

this->UpdateDebuggerPropertyInitializationOffset(sym->GetLocation(), sym->GetPosition());

}

}

}

if (fLabelDefined)

{

this->m\_writer.MarkLabel(doneLabel);

}

if (scopeAcquired)

{

funcInfo->ReleaseTmpRegister(scopeLocation);

}

}

Js::OpCode

ByteCodeGenerator::GetLdSlotOp(Scope \*scope, int envIndex, Js::RegSlot scopeLocation, FuncInfo \*funcInfo)

{

Js::OpCode op;

if (envIndex != -1)

{

if (scope->GetIsObject())

{

op = Js::OpCode::LdEnvObjSlot;

}

else

{

op = Js::OpCode::LdEnvSlot;

}

}

else if (scopeLocation != Js::Constants::NoRegister &&

scopeLocation == funcInfo->frameSlotsRegister)

{

op = Js::OpCode::LdLocalSlot;

}

else if (scopeLocation != Js::Constants::NoRegister &&

scopeLocation == funcInfo->frameObjRegister)

{

op = Js::OpCode::LdLocalObjSlot;

}

else if (scope->HasInnerScopeIndex())

{

if (scope->GetIsObject())

{

op = Js::OpCode::LdInnerObjSlot;

}

else

{

op = Js::OpCode::LdInnerSlot;

}

}

else

{

Assert(scope->GetIsObject());

op = Js::OpCode::LdObjSlot;

}

return op;

}

void ByteCodeGenerator::EmitPropLoad(Js::RegSlot lhsLocation, Symbol \*sym, IdentPtr pid, FuncInfo \*funcInfo)

{

// If sym belongs to a parent frame, get it from the closure environment.

// If it belongs to this func, but there's a non-local reference, get it from the heap-allocated frame.

// (TODO: optimize this by getting the sym from its normal location if there are no non-local defs.)

// Otherwise, just copy the value to the lhsLocation.

Js::ByteCodeLabel doneLabel = 0;

bool fLabelDefined = false;

Js::RegSlot scopeLocation = Js::Constants::NoRegister;

Js::PropertyId envIndex = -1;

Scope \*scope = nullptr;

Scope \*symScope = sym ? sym->GetScope() : this->globalScope;

Assert(symScope);

for (;;)

{

scope = this->FindScopeForSym(symScope, scope, &envIndex, funcInfo);

if (scope == this->globalScope)

{

break;

}

scopeLocation = scope->GetLocation();

if (scope == symScope)

{

break;

}

// Found a scope to which the property may have been added.

Assert(scope && scope->GetIsDynamic());

if (!fLabelDefined)

{

fLabelDefined = true;

doneLabel = this->m\_writer.DefineLabel();

}

Js::ByteCodeLabel nextLabel = this->m\_writer.DefineLabel();

Js::PropertyId propertyId = sym ? sym->EnsurePosition(this) : pid->GetPropertyId();

Js::RegSlot unwrappedScopeLocation = Js::Constants::NoRegister;

bool unwrapWithObj = scope->GetScopeType() == ScopeType\_With && scriptContext->GetConfig()->IsES6UnscopablesEnabled();

if (envIndex != -1)

{

this->m\_writer.BrEnvProperty(

Js::OpCode::BrOnNoEnvProperty,

nextLabel,

funcInfo->FindOrAddReferencedPropertyId(propertyId),

envIndex + Js::FrameDisplay::GetOffsetOfScopes()/sizeof(Js::Var));

Js::RegSlot instLocation = funcInfo->AcquireTmpRegister();

this->m\_writer.SlotI1(

Js::OpCode::LdEnvObj,

instLocation,

envIndex + Js::FrameDisplay::GetOffsetOfScopes()/sizeof(Js::Var));

if (unwrapWithObj)

{

this->m\_writer.Reg2(Js::OpCode::UnwrapWithObj, instLocation, instLocation);

}

this->m\_writer.PatchableProperty(

Js::OpCode::LdFld,

lhsLocation,

instLocation,

funcInfo->FindOrAddInlineCacheId(instLocation, propertyId, false, false));

funcInfo->ReleaseTmpRegister(instLocation);

}

else if (scopeLocation != Js::Constants::NoRegister && scopeLocation == funcInfo->frameObjRegister)

{

this->m\_writer.BrLocalProperty(Js::OpCode::BrOnNoLocalProperty, nextLabel,

funcInfo->FindOrAddReferencedPropertyId(propertyId));

Assert(!unwrapWithObj);

this->m\_writer.ElementP(Js::OpCode::LdLocalFld, lhsLocation,

funcInfo->FindOrAddInlineCacheId(scopeLocation, propertyId, false, false));

}

else

{

this->m\_writer.BrProperty(Js::OpCode::BrOnNoProperty, nextLabel, scopeLocation,

funcInfo->FindOrAddReferencedPropertyId(propertyId));

if (unwrapWithObj)

{

unwrappedScopeLocation = funcInfo->AcquireTmpRegister();

this->m\_writer.Reg2(Js::OpCode::UnwrapWithObj, unwrappedScopeLocation, scopeLocation);

scopeLocation = unwrappedScopeLocation;

}

uint cacheId = funcInfo->FindOrAddInlineCacheId(scopeLocation, propertyId, false, false);

this->m\_writer.PatchableProperty(Js::OpCode::LdFld, lhsLocation, scopeLocation, cacheId);

if (unwrapWithObj)

{

funcInfo->ReleaseTmpRegister(unwrappedScopeLocation);

}

}

this->m\_writer.Br(doneLabel);

this->m\_writer.MarkLabel(nextLabel);

}

// Arrived at the scope in which the property was defined.

if (sym && sym->GetNeedDeclaration() && scope->GetFunc() == funcInfo)

{

// Ensure this symbol has a slot if it needs one.

if (sym->IsInSlot(funcInfo))

{

Js::PropertyId slot = sym->EnsureScopeSlot(funcInfo);

funcInfo->FindOrAddSlotProfileId(scope, slot);

}

EmitUseBeforeDeclarationRuntimeError(this, lhsLocation);

}

else if (sym == nullptr || sym->GetIsGlobal())

{

Js::PropertyId propertyId = sym ? sym->EnsurePosition(this) : pid->GetPropertyId();

if (this->flags & fscrEval)

{

if (funcInfo->byteCodeFunction->GetIsStrictMode() && funcInfo->IsGlobalFunction())

{

uint cacheId = funcInfo->FindOrAddInlineCacheId(funcInfo->frameDisplayRegister, propertyId, false, false);

this->m\_writer.ElementP(Js::OpCode::ScopedLdFld, lhsLocation, cacheId);

}

else

{

uint cacheId = funcInfo->FindOrAddInlineCacheId(funcInfo->GetEnvRegister(), propertyId, false, false);

// Load of a symbol with unknown scope from within eval

// Get it from the closure environment.

this->m\_writer.ElementP(Js::OpCode::ScopedLdFld, lhsLocation, cacheId);

}

}

else if (this->flags & (fscrImplicitThis | fscrImplicitParents))

{

uint cacheId = funcInfo->FindOrAddInlineCacheId(funcInfo->GetEnvRegister(), propertyId, false, false);

// Load of a symbol with unknown scope from within eval or event handler.

// Get it from the closure environment.

this->m\_writer.ElementP(Js::OpCode::ScopedLdFld, lhsLocation, cacheId);

}

else

{

// Special case non-writable built-ins

// TODO: support non-writable global property in general by detecting what attribute the property have current?

// But can't be done if we are byte code serialized, because the attribute might be different for use fields

// next time we run. May want to catch that in the JIT.

Js::OpCode opcode = Js::OpCode::LdRootFld;

// These properties are non-writable

switch (propertyId)

{

case Js::PropertyIds::NaN:

opcode = Js::OpCode::LdNaN;

break;

case Js::PropertyIds::Infinity:

opcode = Js::OpCode::LdInfinity;

break;

case Js::PropertyIds::undefined:

opcode = Js::OpCode::LdUndef;

break;

}

if (opcode == Js::OpCode::LdRootFld)

{

this->EmitPatchableRootProperty(Js::OpCode::LdRootFld, lhsLocation, propertyId, false, false, funcInfo);

}

else

{

this->Writer()->Reg1(opcode, lhsLocation);

}

}

}

else if (sym->IsInSlot(funcInfo) || envIndex != -1)

{

// Make sure the property has a slot. This will bump up the size of the slot array if necessary.

Js::PropertyId slot = sym->EnsureScopeSlot(funcInfo);

Js::ProfileId profileId = funcInfo->FindOrAddSlotProfileId(scope, slot);

bool chkBlockVar = NeedCheckBlockVar(sym, scope, funcInfo);

Js::OpCode op;

// Now get the property from its slot.

op = this->GetLdSlotOp(scope, envIndex, scopeLocation, funcInfo);

slot = slot + (sym->GetScope()->GetIsObject() ? 0 : Js::ScopeSlots::FirstSlotIndex);

if (envIndex != -1)

{

this->m\_writer.SlotI2(op, lhsLocation, envIndex + Js::FrameDisplay::GetOffsetOfScopes()/sizeof(Js::Var), slot, profileId);

}

else if (scopeLocation != Js::Constants::NoRegister &&

(scopeLocation == funcInfo->frameSlotsRegister || scopeLocation == funcInfo->frameObjRegister))

{

this->m\_writer.SlotI1(op, lhsLocation, slot, profileId);

}

else if (scope->HasInnerScopeIndex())

{

this->m\_writer.SlotI2(op, lhsLocation, scope->GetInnerScopeIndex(), slot, profileId);

}

else

{

Assert(scope->GetIsObject());

this->m\_writer.Slot(op, lhsLocation, scopeLocation, slot, profileId);

}

if (chkBlockVar)

{

this->m\_writer.Reg1(Js::OpCode::ChkUndecl, lhsLocation);

}

}

else

{

if (lhsLocation != sym->GetLocation())

{

this->m\_writer.Reg2(Js::OpCode::Ld\_A, lhsLocation, sym->GetLocation());

}

if (sym->GetIsBlockVar() && ((sym->GetDecl()->nop == knopLetDecl || sym->GetDecl()->nop == knopConstDecl) && sym->GetDecl()->sxVar.isSwitchStmtDecl) && CONFIG\_FLAG(TDZ))

{

this->m\_writer.Reg1(Js::OpCode::ChkUndecl, lhsLocation);

}

}

if (fLabelDefined)

{

this->m\_writer.MarkLabel(doneLabel);

}

}

bool ByteCodeGenerator::NeedCheckBlockVar(Symbol\* sym, Scope\* scope, FuncInfo\* funcInfo) const

{

bool tdz = sym->GetIsBlockVar()

&& (scope->GetFunc() != funcInfo || ((sym->GetDecl()->nop == knopLetDecl || sym->GetDecl()->nop == knopConstDecl) && sym->GetDecl()->sxVar.isSwitchStmtDecl))

&& CONFIG\_FLAG(TDZ);

return tdz || sym->GetIsNonSimpleParameter();

}

void ByteCodeGenerator::EmitPropDelete(Js::RegSlot lhsLocation, Symbol \*sym, IdentPtr pid, FuncInfo \*funcInfo)

{

// If sym belongs to a parent frame, delete it from the closure environment.

// If it belongs to this func, but there's a non-local reference, get it from the heap-allocated frame.

// (TODO: optimize this by getting the sym from its normal location if there are no non-local defs.)

// Otherwise, just return false.

Js::ByteCodeLabel doneLabel = 0;

bool fLabelDefined = false;

Js::RegSlot scopeLocation = Js::Constants::NoRegister;

Js::PropertyId envIndex = -1;

Scope \*scope = nullptr;

Scope \*symScope = sym ? sym->GetScope() : this->globalScope;

Assert(symScope);

for (;;)

{

scope = this->FindScopeForSym(symScope, scope, &envIndex, funcInfo);

if (scope == this->globalScope)

{

scopeLocation = ByteCodeGenerator::RootObjectRegister;

}

else if (envIndex == -1)

{

Assert(funcInfo == scope->GetFunc());

scopeLocation = scope->GetLocation();

}

if (scope == symScope)

{

break;

}

// Found a scope to which the property may have been added.

Assert(scope && scope->GetIsDynamic());

if (!fLabelDefined)

{

fLabelDefined = true;

doneLabel = this->m\_writer.DefineLabel();

}

Js::ByteCodeLabel nextLabel = this->m\_writer.DefineLabel();

Js::PropertyId propertyId = sym ? sym->EnsurePosition(this) : pid->GetPropertyId();

bool unwrapWithObj = scope->GetScopeType() == ScopeType\_With && scriptContext->GetConfig()->IsES6UnscopablesEnabled();

if (envIndex != -1)

{

this->m\_writer.BrEnvProperty(

Js::OpCode::BrOnNoEnvProperty,

nextLabel,

funcInfo->FindOrAddReferencedPropertyId(propertyId),

envIndex + Js::FrameDisplay::GetOffsetOfScopes()/sizeof(Js::Var));

Js::RegSlot instLocation = funcInfo->AcquireTmpRegister();

this->m\_writer.SlotI1(

Js::OpCode::LdEnvObj,

instLocation,

envIndex + Js::FrameDisplay::GetOffsetOfScopes()/sizeof(Js::Var));

if (unwrapWithObj)

{

this->m\_writer.Reg2(Js::OpCode::UnwrapWithObj, instLocation, instLocation);

}

this->m\_writer.Property(Js::OpCode::DeleteFld, lhsLocation, instLocation,

funcInfo->FindOrAddReferencedPropertyId(propertyId));

funcInfo->ReleaseTmpRegister(instLocation);

}

else if (scopeLocation != Js::Constants::NoRegister && scopeLocation == funcInfo->frameObjRegister)

{

this->m\_writer.BrLocalProperty(Js::OpCode::BrOnNoLocalProperty, nextLabel,

funcInfo->FindOrAddReferencedPropertyId(propertyId));

Assert(!unwrapWithObj);

this->m\_writer.ElementU(Js::OpCode::DeleteLocalFld, lhsLocation,

funcInfo->FindOrAddReferencedPropertyId(propertyId));

}

else

{

this->m\_writer.BrProperty(Js::OpCode::BrOnNoProperty, nextLabel, scopeLocation,

funcInfo->FindOrAddReferencedPropertyId(propertyId));

Js::RegSlot unwrappedScopeLocation = Js::Constants::NoRegister;

if (unwrapWithObj)

{

unwrappedScopeLocation = funcInfo->AcquireTmpRegister();

this->m\_writer.Reg2(Js::OpCode::UnwrapWithObj, unwrappedScopeLocation, scopeLocation);

scopeLocation = unwrappedScopeLocation;

}

this->m\_writer.Property(Js::OpCode::DeleteFld, lhsLocation, scopeLocation,

funcInfo->FindOrAddReferencedPropertyId(propertyId));

if (unwrapWithObj)

{

funcInfo->ReleaseTmpRegister(unwrappedScopeLocation);

}

}

this->m\_writer.Br(doneLabel);

this->m\_writer.MarkLabel(nextLabel);

}

// Arrived at the scope in which the property was defined.

if (sym == nullptr || sym->GetIsGlobal())

{

Js::PropertyId propertyId = sym ? sym->EnsurePosition(this) : pid->GetPropertyId();

if (this->flags & (fscrEval | fscrImplicitThis | fscrImplicitParents))

{

this->m\_writer.ScopedProperty(Js::OpCode::ScopedDeleteFld, lhsLocation,

funcInfo->FindOrAddReferencedPropertyId(propertyId));

}

else

{

this->m\_writer.Property(Js::OpCode::DeleteRootFld, lhsLocation, ByteCodeGenerator::RootObjectRegister,

funcInfo->FindOrAddReferencedPropertyId(propertyId));

}

}

else

{

// The delete will look like a non-local reference, so make sure a slot is reserved.

sym->EnsureScopeSlot(funcInfo);

this->m\_writer.Reg1(Js::OpCode::LdFalse, lhsLocation);

}

if (fLabelDefined)

{

this->m\_writer.MarkLabel(doneLabel);

}

}

void ByteCodeGenerator::EmitTypeOfFld(FuncInfo \* funcInfo, Js::PropertyId propertyId, Js::RegSlot value, Js::RegSlot instance, Js::OpCode ldFldOp)

{

uint cacheId;

Js::RegSlot tmpReg = funcInfo->AcquireTmpRegister();

switch (ldFldOp)

{

case Js::OpCode::LdRootFldForTypeOf:

cacheId = funcInfo->FindOrAddRootObjectInlineCacheId(propertyId, false, false);

this->Writer()->PatchableRootProperty(ldFldOp, tmpReg, cacheId, false, false);

break;

case Js::OpCode::LdLocalFld:

case Js::OpCode::ScopedLdFldForTypeOf:

cacheId = funcInfo->FindOrAddInlineCacheId(instance, propertyId, false, false);

this->Writer()->ElementP(ldFldOp, tmpReg, cacheId);

break;

default:

cacheId = funcInfo->FindOrAddInlineCacheId(instance, propertyId, false, false);

this->Writer()->PatchableProperty(ldFldOp, tmpReg, instance, cacheId);

break;

}

this->Writer()->Reg2(Js::OpCode::Typeof, value, tmpReg);

funcInfo->ReleaseTmpRegister(tmpReg);

}

void ByteCodeGenerator::EmitPropTypeof(Js::RegSlot lhsLocation, Symbol \*sym, IdentPtr pid, FuncInfo \*funcInfo)

{

// If sym belongs to a parent frame, delete it from the closure environment.

// If it belongs to this func, but there's a non-local reference, get it from the heap-allocated frame.

// (TODO: optimize this by getting the sym from its normal location if there are no non-local defs.)

// Otherwise, just return false

Js::ByteCodeLabel doneLabel = 0;

bool fLabelDefined = false;

Js::RegSlot scopeLocation = Js::Constants::NoRegister;

Js::PropertyId envIndex = -1;

Scope \*scope = nullptr;

Scope \*symScope = sym ? sym->GetScope() : this->globalScope;

Assert(symScope);

for (;;)

{

scope = this->FindScopeForSym(symScope, scope, &envIndex, funcInfo);

if (scope == this->globalScope)

{

scopeLocation = ByteCodeGenerator::RootObjectRegister;

}

else if (envIndex == -1)

{

Assert(funcInfo == scope->GetFunc());

scopeLocation = scope->GetLocation();

}

if (scope == symScope)

{

break;

}

// Found a scope to which the property may have been added.

Assert(scope && scope->GetIsDynamic());

if (!fLabelDefined)

{

fLabelDefined = true;

doneLabel = this->m\_writer.DefineLabel();

}

Js::ByteCodeLabel nextLabel = this->m\_writer.DefineLabel();

Js::PropertyId propertyId = sym ? sym->EnsurePosition(this) : pid->GetPropertyId();

bool unwrapWithObj = scope->GetScopeType() == ScopeType\_With && scriptContext->GetConfig()->IsES6UnscopablesEnabled();

if (envIndex != -1)

{

this->m\_writer.BrEnvProperty(Js::OpCode::BrOnNoEnvProperty, nextLabel,

funcInfo->FindOrAddReferencedPropertyId(propertyId),

envIndex + Js::FrameDisplay::GetOffsetOfScopes()/sizeof(Js::Var));

Js::RegSlot instLocation = funcInfo->AcquireTmpRegister();

this->m\_writer.SlotI1(Js::OpCode::LdEnvObj,

instLocation,

envIndex + Js::FrameDisplay::GetOffsetOfScopes()/sizeof(Js::Var));

if (unwrapWithObj)

{

this->m\_writer.Reg2(Js::OpCode::UnwrapWithObj, instLocation, instLocation);

}

this->EmitTypeOfFld(funcInfo, propertyId, lhsLocation, instLocation, Js::OpCode::LdFldForTypeOf);

funcInfo->ReleaseTmpRegister(instLocation);

}

else if (scopeLocation != Js::Constants::NoRegister && scopeLocation == funcInfo->frameObjRegister)

{

this->m\_writer.BrLocalProperty(Js::OpCode::BrOnNoLocalProperty, nextLabel,

funcInfo->FindOrAddReferencedPropertyId(propertyId));

Assert(!unwrapWithObj);

this->EmitTypeOfFld(funcInfo, propertyId, lhsLocation, scopeLocation, Js::OpCode::LdLocalFld);

}

else

{

this->m\_writer.BrProperty(Js::OpCode::BrOnNoProperty, nextLabel, scopeLocation,

funcInfo->FindOrAddReferencedPropertyId(propertyId));

Js::RegSlot unwrappedScopeLocation = Js::Constants::NoRegister;

if (unwrapWithObj)

{

unwrappedScopeLocation = funcInfo->AcquireTmpRegister();

this->m\_writer.Reg2(Js::OpCode::UnwrapWithObj, unwrappedScopeLocation, scopeLocation);

scopeLocation = unwrappedScopeLocation;

}

this->EmitTypeOfFld(funcInfo, propertyId, lhsLocation, scopeLocation, Js::OpCode::LdFldForTypeOf);

if (unwrapWithObj)

{

funcInfo->ReleaseTmpRegister(unwrappedScopeLocation);

}

}

this->m\_writer.Br(doneLabel);

this->m\_writer.MarkLabel(nextLabel);

}

// Arrived at the scope in which the property was defined.

if (sym && sym->GetNeedDeclaration() && scope->GetFunc() == funcInfo)

{

// Ensure this symbol has a slot if it needs one.

if (sym->IsInSlot(funcInfo))

{

Js::PropertyId slot = sym->EnsureScopeSlot(funcInfo);

funcInfo->FindOrAddSlotProfileId(scope, slot);

}

EmitUseBeforeDeclarationRuntimeError(this, lhsLocation);

}

else if (sym == nullptr || sym->GetIsGlobal())

{

Js::PropertyId propertyId = sym ? sym->EnsurePosition(this) : pid->GetPropertyId();

if (this->flags & fscrEval)

{

if (funcInfo->byteCodeFunction->GetIsStrictMode() && funcInfo->IsGlobalFunction())

{

this->EmitTypeOfFld(funcInfo, propertyId, lhsLocation, funcInfo->frameDisplayRegister, Js::OpCode::ScopedLdFldForTypeOf);

}

else

{

this->EmitTypeOfFld(funcInfo, propertyId, lhsLocation, funcInfo->GetEnvRegister(), Js::OpCode::ScopedLdFldForTypeOf);

}

}

else if (this->flags & (fscrImplicitThis | fscrImplicitParents))

{

this->EmitTypeOfFld(funcInfo, propertyId, lhsLocation, funcInfo->GetEnvRegister(), Js::OpCode::ScopedLdFldForTypeOf);

}

else

{

this->EmitTypeOfFld(funcInfo, propertyId, lhsLocation, ByteCodeGenerator::RootObjectRegister, Js::OpCode::LdRootFldForTypeOf);

}

}

else if (sym->IsInSlot(funcInfo) || envIndex != -1)

{

// Make sure the property has a slot. This will bump up the size of the slot array if necessary.

Js::PropertyId slot = sym->EnsureScopeSlot(funcInfo);

Js::ProfileId profileId = funcInfo->FindOrAddSlotProfileId(scope, slot);

Js::RegSlot tmpLocation = funcInfo->AcquireTmpRegister();

bool chkBlockVar = NeedCheckBlockVar(sym, scope, funcInfo);

Js::OpCode op;

op = this->GetLdSlotOp(scope, envIndex, scopeLocation, funcInfo);

slot = slot + (sym->GetScope()->GetIsObject() ? 0 : Js::ScopeSlots::FirstSlotIndex);

if (envIndex != -1)

{

this->m\_writer.SlotI2(op, tmpLocation, envIndex + Js::FrameDisplay::GetOffsetOfScopes()/sizeof(Js::Var), slot, profileId);

}

else if (scopeLocation != Js::Constants::NoRegister &&

(scopeLocation == funcInfo->frameSlotsRegister || scopeLocation == funcInfo->frameObjRegister))

{

this->m\_writer.SlotI1(op, tmpLocation, slot, profileId);

}

else if (scope->HasInnerScopeIndex())

{

this->m\_writer.SlotI2(op, tmpLocation, scope->GetInnerScopeIndex(), slot, profileId);

}

else

{

Assert(scope->GetIsObject());

this->m\_writer.Slot(op, tmpLocation, scopeLocation, slot, profileId);

}

if (chkBlockVar)

{

this->m\_writer.Reg1(Js::OpCode::ChkUndecl, tmpLocation);

}

this->m\_writer.Reg2(Js::OpCode::Typeof, lhsLocation, tmpLocation);

funcInfo->ReleaseTmpRegister(tmpLocation);

}

else

{

this->m\_writer.Reg2(Js::OpCode::Typeof, lhsLocation, sym->GetLocation());

}

if (fLabelDefined)

{

this->m\_writer.MarkLabel(doneLabel);

}

}

void ByteCodeGenerator::EnsureNoRedeclarations(ParseNode \*pnodeBlock, FuncInfo \*funcInfo)

{

// Emit dynamic runtime checks for variable re-declarations. Only necessary for global functions (script or eval).

// In eval only var declarations can cause redeclaration, and only in non-strict mode, because let/const variables

// remain local to the eval code.

Assert(pnodeBlock->nop == knopBlock);

Assert(pnodeBlock->sxBlock.blockType == PnodeBlockType::Global || pnodeBlock->sxBlock.scope->GetScopeType() == ScopeType\_GlobalEvalBlock);

if (!(this->flags & fscrEvalCode))

{

IterateBlockScopedVariables(pnodeBlock, [this](ParseNode \*pnode)

{

FuncInfo \*funcInfo = this->TopFuncInfo();

Symbol \*sym = pnode->sxVar.sym;

Assert(sym->GetIsGlobal());

Js::PropertyId propertyId = sym->EnsurePosition(this);

this->m\_writer.ElementRootU(Js::OpCode::EnsureNoRootFld, funcInfo->FindOrAddReferencedPropertyId(propertyId));

});

}

for (ParseNode \*pnode = funcInfo->root->sxFnc.pnodeVars; pnode; pnode = pnode->sxVar.pnodeNext)

{

Symbol\* sym = pnode->sxVar.sym;

if (sym == nullptr || pnode->sxVar.isBlockScopeFncDeclVar)

continue;

if (sym->GetIsCatch() || (pnode->nop == knopVarDecl && sym->GetIsBlockVar()))

{

// The init node was bound to the catch object, because it's inside a catch and has the

// same name as the catch object. But we want to define a user var at function scope,

// so find the right symbol. (We'll still assign the RHS value to the catch object symbol.)

// This also applies to a var declaration in the same scope as a let declaration.

// Assert that catch cannot be at function scope and let and var at function scope is redeclaration error.

Assert(sym->GetIsCatch() || funcInfo->bodyScope != sym->GetScope() || !this->scriptContext->GetConfig()->IsBlockScopeEnabled());

sym = funcInfo->bodyScope->FindLocalSymbol(sym->GetName());

Assert(sym && !sym->GetIsCatch() && !sym->GetIsBlockVar());

}

Assert(sym->GetIsGlobal());

if (sym->GetSymbolType() == STVariable)

{

Js::PropertyId propertyId = sym->EnsurePosition(this);

if (this->flags & fscrEval)

{

if (!funcInfo->byteCodeFunction->GetIsStrictMode())

{

this->m\_writer.ScopedProperty(Js::OpCode::ScopedEnsureNoRedeclFld, ByteCodeGenerator::RootObjectRegister,

funcInfo->FindOrAddReferencedPropertyId(propertyId));

}

}

else

{

this->m\_writer.ElementRootU(Js::OpCode::EnsureNoRootRedeclFld, funcInfo->FindOrAddReferencedPropertyId(propertyId));

}

}

}

}

void ByteCodeGenerator::RecordAllIntConstants(FuncInfo \* funcInfo)

{

Js::FunctionBody \*byteCodeFunction = this->TopFuncInfo()->GetParsedFunctionBody();

funcInfo->constantToRegister.Map([byteCodeFunction](unsigned int val, Js::RegSlot location)

{

byteCodeFunction->RecordIntConstant(byteCodeFunction->MapRegSlot(location), val);

});

}

void ByteCodeGenerator::RecordAllStrConstants(FuncInfo \* funcInfo)

{

Js::FunctionBody \*byteCodeFunction = this->TopFuncInfo()->GetParsedFunctionBody();

funcInfo->stringToRegister.Map([byteCodeFunction](IdentPtr pid, Js::RegSlot location)

{

byteCodeFunction->RecordStrConstant(byteCodeFunction->MapRegSlot(location), pid->Psz(), pid->Cch());

});

}

void ByteCodeGenerator::RecordAllStringTemplateCallsiteConstants(FuncInfo\* funcInfo)

{

Js::FunctionBody \*byteCodeFunction = this->TopFuncInfo()->GetParsedFunctionBody();

funcInfo->stringTemplateCallsiteRegisterMap.Map([byteCodeFunction](ParseNodePtr pnode, Js::RegSlot location)

{

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

Js::JavascriptLibrary\* library = scriptContext->GetLibrary();

Js::RecyclableObject\* callsiteObject = library->TryGetStringTemplateCallsiteObject(pnode);

if (callsiteObject == nullptr)

{

Js::RecyclableObject\* rawArray = ByteCodeGenerator::BuildArrayFromStringList(pnode->sxStrTemplate.pnodeStringRawLiterals, pnode->sxStrTemplate.countStringLiterals, scriptContext);

rawArray->Freeze();

callsiteObject = ByteCodeGenerator::BuildArrayFromStringList(pnode->sxStrTemplate.pnodeStringLiterals, pnode->sxStrTemplate.countStringLiterals, scriptContext);

callsiteObject->SetPropertyWithAttributes(Js::PropertyIds::raw, rawArray, PropertyNone, nullptr);

callsiteObject->Freeze();

library->AddStringTemplateCallsiteObject(callsiteObject);

}

byteCodeFunction->RecordConstant(byteCodeFunction->MapRegSlot(location), callsiteObject);

});

}

bool IsApplyArgs(ParseNode\* callNode)

{

ParseNode\* target = callNode->sxCall.pnodeTarget;

ParseNode\* args = callNode->sxCall.pnodeArgs;

if ((target != nullptr) && (target->nop == knopDot))

{

ParseNode\* lhsNode = target->sxBin.pnode1;

if ((lhsNode != nullptr) && ((lhsNode->nop == knopDot) || (lhsNode->nop == knopName)) && !IsArguments(lhsNode))

{

ParseNode\* nameNode = target->sxBin.pnode2;

if (nameNode != nullptr)

{

bool nameIsApply = nameNode->sxPid.PropertyIdFromNameNode() == Js::PropertyIds::apply;

if (nameIsApply && args != nullptr && args->nop == knopList)

{

ParseNode\* arg1 = args->sxBin.pnode1;

ParseNode\* arg2 = args->sxBin.pnode2;

if ((arg1 != nullptr) && (arg1->nop == knopThis) && (arg2 != nullptr) && (arg2->nop == knopName) && (arg2->sxPid.sym != nullptr))

{

return arg2->sxPid.sym->GetIsArguments();

}

}

}

}

}

return false;

}

void PostCheckApplyEnclosesArgs(ParseNode\* pnode, ByteCodeGenerator\* byteCodeGenerator, ApplyCheck\* applyCheck)

{

if ((pnode == nullptr) || (!applyCheck->matches))

{

return;

}

if (pnode->nop == knopCall)

{

if ((!pnode->isUsed) && IsApplyArgs(pnode))

{

if (!applyCheck->insideApplyCall)

{

applyCheck->matches = false;

}

applyCheck->insideApplyCall = false;

}

}

}

void CheckApplyEnclosesArgs(ParseNode\* pnode, ByteCodeGenerator\* byteCodeGenerator, ApplyCheck\* applyCheck)

{

if ((pnode == nullptr) || (!applyCheck->matches))

{

return;

}

switch (pnode->nop)

{

case knopName:

{

Symbol\* sym = pnode->sxPid.sym;

if (sym != nullptr)

{

if (sym->GetIsArguments())

{

if (!applyCheck->insideApplyCall)

{

applyCheck->matches = false;

}

}

}

break;

}

case knopCall:

if ((!pnode->isUsed) && IsApplyArgs(pnode))

{

// no nested apply calls

if (applyCheck->insideApplyCall)

{

applyCheck->matches = false;

}

else

{

applyCheck->insideApplyCall = true;

applyCheck->sawApply = true;

pnode->sxCall.isApplyCall = true;

}

}

break;

}

}

unsigned int CountArguments(ParseNode \*pnode, BOOL \*pSideEffect = nullptr)

{

// If the caller passed us a pSideEffect, it wants to know whether there are potential

// side-effects in the argument list. We need to know this so that the call target

// operands can be preserved if necessary.

// For now, treat any non-leaf op as a potential side-effect. This causes no detectable slowdowns,

// but we can be more precise if we need to be.

if (pSideEffect)

{

\*pSideEffect = FALSE;

}

unsigned int argCount = 1;

if (pnode != nullptr)

{

while (pnode->nop == knopList)

{

argCount++;

if (pSideEffect && !(ParseNode::Grfnop(pnode->sxBin.pnode1->nop) & fnopLeaf))

{

\*pSideEffect = TRUE;

}

pnode = pnode->sxBin.pnode2;

}

argCount++;

if (pSideEffect && !(ParseNode::Grfnop(pnode->nop) & fnopLeaf))

{

\*pSideEffect = TRUE;

}

}

return argCount;

}

void SaveOpndValue(ParseNode \*pnode, FuncInfo \*funcInfo)

{

// Save a local name to a register other than its home location.

// This guards against side-effects in cases like x.foo(x = bar()).

Symbol \*sym = nullptr;

if (pnode->nop == knopName)

{

sym = pnode->sxPid.sym;

}

else if (pnode->nop == knopComputedName)

{

ParseNode \*pnode1 = pnode->sxUni.pnode1;

if (pnode1->nop == knopName)

{

sym = pnode1->sxPid.sym;

}

}

if (sym == nullptr)

{

return;

}

// If the target is a local being kept in its home location,

// protect the target's value in the event the home location is overwritten.

if (pnode->location != Js::Constants::NoRegister &&

sym->GetScope()->GetFunc() == funcInfo &&

pnode->location == sym->GetLocation())

{

pnode->location = funcInfo->AcquireTmpRegister();

}

}

void ByteCodeGenerator::StartStatement(ParseNode\* node)

{

Assert(TopFuncInfo() != nullptr);

m\_writer.StartStatement(node, TopFuncInfo()->curTmpReg - TopFuncInfo()->firstTmpReg);

}

void ByteCodeGenerator::EndStatement(ParseNode\* node)

{

m\_writer.EndStatement(node);

}

void ByteCodeGenerator::StartSubexpression(ParseNode\* node)

{

Assert(TopFuncInfo() != nullptr);

m\_writer.StartSubexpression(node);

}

void ByteCodeGenerator::EndSubexpression(ParseNode\* node)

{

m\_writer.EndSubexpression(node);

}

void EmitReference(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo)

{

// Generate code for the LHS of an assignment.

switch (pnode->nop)

{

case knopDot:

Emit(pnode->sxBin.pnode1, byteCodeGenerator, funcInfo, false);

break;

case knopIndex:

Emit(pnode->sxBin.pnode1, byteCodeGenerator, funcInfo, false);

Emit(pnode->sxBin.pnode2, byteCodeGenerator, funcInfo, false);

break;

case knopName:

break;

case knopArrayPattern:

case knopObjectPattern:

break;

case knopCall:

case knopNew:

// Emit the operands of a call that will be used as a LHS.

// These have to be emitted before the RHS, but they have to persist until

// the end of the expression.

// Emit the call target operands first.

switch (pnode->sxCall.pnodeTarget->nop)

{

case knopDot:

case knopIndex:

funcInfo->AcquireLoc(pnode->sxCall.pnodeTarget);

EmitReference(pnode->sxCall.pnodeTarget, byteCodeGenerator, funcInfo);

break;

case knopName:

{

Symbol \*sym = pnode->sxCall.pnodeTarget->sxPid.sym;

if (!sym || sym->GetLocation() == Js::Constants::NoRegister)

{

funcInfo->AcquireLoc(pnode->sxCall.pnodeTarget);

}

if (sym && (sym->IsInSlot(funcInfo) || sym->GetScope()->GetFunc() != funcInfo))

{

// Can't get the value from the assigned register, so load it here.

EmitLoad(pnode->sxCall.pnodeTarget, byteCodeGenerator, funcInfo);

}

else

{

EmitReference(pnode->sxCall.pnodeTarget, byteCodeGenerator, funcInfo);

}

break;

}

default:

EmitLoad(pnode->sxCall.pnodeTarget, byteCodeGenerator, funcInfo);

break;

}

// Now the arg list. We evaluate everything now and emit the ArgOut's later.

if (pnode->sxCall.pnodeArgs)

{

ParseNode \*pnodeArg = pnode->sxCall.pnodeArgs;

while (pnodeArg->nop == knopList)

{

Emit(pnodeArg->sxBin.pnode1, byteCodeGenerator, funcInfo, false);

pnodeArg = pnodeArg->sxBin.pnode2;

}

Emit(pnodeArg, byteCodeGenerator, funcInfo, false);

}

break;

default:

Emit(pnode, byteCodeGenerator, funcInfo, false);

break;

}

}

void EmitGetIterator(Js::RegSlot iteratorLocation, Js::RegSlot iterableLocation, ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* funcInfo);

void EmitIteratorNext(Js::RegSlot itemLocation, Js::RegSlot iteratorLocation, Js::RegSlot nextInputLocation, ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* funcInfo);

void EmitIteratorComplete(Js::RegSlot doneLocation, Js::RegSlot iteratorResultLocation, ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* funcInfo);

void EmitIteratorValue(Js::RegSlot valueLocation, Js::RegSlot iteratorResultLocation, ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* funcInfo);

void EmitDestructuredElement(ParseNode \*elem, Js::RegSlot sourceLocation, ByteCodeGenerator\* byteCodeGenerator, FuncInfo \*funcInfo)

{

switch (elem->nop)

{

case knopVarDecl:

case knopLetDecl:

case knopConstDecl:

// We manually need to set NeedDeclaration since the node won't be visited.

elem->sxVar.sym->SetNeedDeclaration(false);

break;

case knopName:

if (elem->sxPid.sym != nullptr && elem->sxPid.sym->GetNeedDeclaration())

{

bool needToRelease = elem->location == Js::Constants::NoRegister;

EmitUseBeforeDeclaration(elem, byteCodeGenerator, funcInfo);

if (needToRelease && elem->location != Js::Constants::NoRegister)

{

// We have acquired register as a part of EmitUseBeforeDeclaration. We need to release that.

funcInfo->ReleaseTmpRegister(elem->location);

}

}

break;

default:

EmitReference(elem, byteCodeGenerator, funcInfo);

}

EmitAssignment(nullptr, elem, sourceLocation, byteCodeGenerator, funcInfo);

funcInfo->ReleaseReference(elem);

}

void EmitDestructuredRestArray(ParseNode \*elem, Js::RegSlot iteratorLocation, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo)

{

Js::RegSlot restArrayLocation = funcInfo->AcquireTmpRegister();

byteCodeGenerator->Writer()->Reg1Unsigned1(

Js::OpCode::NewScArray,

restArrayLocation,

ByteCodeGenerator::DefaultArraySize);

// BytecodeGen can't convey to IRBuilder that some of the temporaries used here are live. When we

// have a rest parameter, a counter is used in a loop for the array index, but there is no way to

// convey this is live on the back edge.

// As a workaround, we have a persistent var reg that is used for the loop counter

Js::RegSlot counterLocation = elem->location;

// TODO[ianhall]: Is calling EnregisterConstant() during Emit phase allowed?

Js::RegSlot zeroConstantReg = byteCodeGenerator->EnregisterConstant(0);

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, counterLocation, zeroConstantReg);

// loopTop:

Js::ByteCodeLabel loopTop = byteCodeGenerator->Writer()->DefineLabel();

byteCodeGenerator->Writer()->MarkLabel(loopTop);

Js::RegSlot itemLocation = funcInfo->AcquireTmpRegister();

EmitIteratorNext(itemLocation, iteratorLocation, Js::Constants::NoRegister, byteCodeGenerator, funcInfo);

Js::RegSlot doneLocation = funcInfo->AcquireTmpRegister();

EmitIteratorComplete(doneLocation, itemLocation, byteCodeGenerator, funcInfo);

Js::ByteCodeLabel iteratorDone = byteCodeGenerator->Writer()->DefineLabel();

byteCodeGenerator->Writer()->BrReg1(Js::OpCode::BrTrue\_A, iteratorDone, doneLocation);

Js::RegSlot valueLocation = funcInfo->AcquireTmpRegister();

EmitIteratorValue(valueLocation, itemLocation, byteCodeGenerator, funcInfo);

byteCodeGenerator->Writer()->Element(

ByteCodeGenerator::GetStElemIOpCode(funcInfo),

valueLocation, restArrayLocation, counterLocation);

funcInfo->ReleaseTmpRegister(valueLocation);

funcInfo->ReleaseTmpRegister(doneLocation);

funcInfo->ReleaseTmpRegister(itemLocation);

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Incr\_A, counterLocation, counterLocation);

byteCodeGenerator->Writer()->Br(loopTop);

// iteratorDone:

byteCodeGenerator->Writer()->MarkLabel(iteratorDone);

ParseNode \*restElem = elem->sxUni.pnode1;

EmitDestructuredElement(restElem, restArrayLocation, byteCodeGenerator, funcInfo);

funcInfo->ReleaseTmpRegister(restArrayLocation);

}

/\*

EmitDestructuredArray(lhsArray, rhs):

iterator = rhs[@@iterator]

if lhsArray empty

return

for each element in lhsArray except rest

value = iterator.next()

if element is a nested destructured array

EmitDestructuredArray(element, value)

else

if value is undefined and there is an initializer

evaluate initializer

evaluate element reference

element = initializer

else

element = value

if lhsArray has a rest element

rest = []

while iterator is not done

value = iterator.next()

rest.append(value)

\*/

void EmitDestructuredArray(

ParseNode \*lhs,

Js::RegSlot rhsLocation,

ByteCodeGenerator \*byteCodeGenerator,

FuncInfo \*funcInfo)

{

byteCodeGenerator->StartStatement(lhs);

Js::RegSlot iteratorLocation = funcInfo->AcquireTmpRegister();

EmitGetIterator(iteratorLocation, rhsLocation, byteCodeGenerator, funcInfo);

Assert(lhs->nop == knopArrayPattern);

ParseNode \*list = lhs->sxArrLit.pnode1;

if (list == nullptr)

{

// No elements to bind or assign.

funcInfo->ReleaseTmpRegister(iteratorLocation);

return;

}

ParseNode \*elem = nullptr;

while (list != nullptr)

{

ParseNode \*init = nullptr;

if (list->nop == knopList)

{

elem = list->sxBin.pnode1;

}

else

{

elem = list;

}

if (elem->nop == knopEllipsis)

{

// Rest must be the last argument - no need to continue.

break;

}

switch (elem->nop)

{

case knopAsg:

// An assignment node will always have an initializer

init = elem->sxBin.pnode2;

elem = elem->sxBin.pnode1;

break;

case knopVarDecl:

case knopLetDecl:

case knopConstDecl:

init = elem->sxVar.pnodeInit;

break;

default:

break;

}

byteCodeGenerator->StartStatement(elem);

Js::RegSlot itemLocation = funcInfo->AcquireTmpRegister();

EmitIteratorNext(itemLocation, iteratorLocation, Js::Constants::NoRegister, byteCodeGenerator, funcInfo);

if (elem->nop == knopEmpty)

{

// Missing elements only require a next call.

funcInfo->ReleaseTmpRegister(itemLocation);

if (list->nop == knopList)

{

list = list->sxBin.pnode2;

continue;

}

else

{

break;

}

}

Js::RegSlot doneLocation = funcInfo->AcquireTmpRegister();

EmitIteratorComplete(doneLocation, itemLocation, byteCodeGenerator, funcInfo);

// If the iterator hasn't completed, skip assigning undefined.

Js::ByteCodeLabel iteratorAlreadyDone = byteCodeGenerator->Writer()->DefineLabel();

byteCodeGenerator->Writer()->BrReg1(Js::OpCode::BrTrue\_A, iteratorAlreadyDone, doneLocation);

funcInfo->ReleaseTmpRegister(doneLocation);

// We're not done with the iterator, so assign the .next() value.

Js::RegSlot valueLocation = funcInfo->AcquireTmpRegister();

EmitIteratorValue(valueLocation, itemLocation, byteCodeGenerator, funcInfo);

Js::ByteCodeLabel beforeDefaultAssign = byteCodeGenerator->Writer()->DefineLabel();

byteCodeGenerator->Writer()->Br(beforeDefaultAssign);

// iteratorAlreadyDone:

byteCodeGenerator->Writer()->MarkLabel(iteratorAlreadyDone);

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, valueLocation, funcInfo->undefinedConstantRegister);

// beforeDefaultAssign:

byteCodeGenerator->Writer()->MarkLabel(beforeDefaultAssign);

if (elem->IsPattern())

{

// If we get an undefined value and have an initializer, use it in place of undefined.

if (init != nullptr)

{

/\*

the IR builder uses two symbols for a temp register in the if else path

R9 <- R3

if (...)

R9 <- R2

R10 = R9.<property> // error -> IR creates a new lifetime for the if path, and the direct path dest is not referenced

hence we have to create a new temp

TEMP REG USED TO FIX THIS PRODUCES THIS

R9 <- R3

if (BrEq\_A R9, R3)

R10 <- R2 :

else

R10 <- R9 : skipdefault

... = R10[@@iterator] : loadIter

\*/

// Temp Register

Js::RegSlot valueLocationTmp = funcInfo->AcquireTmpRegister();

byteCodeGenerator->StartStatement(init);

Js::ByteCodeLabel skipDefault = byteCodeGenerator->Writer()->DefineLabel();

Js::ByteCodeLabel loadIter = byteCodeGenerator->Writer()->DefineLabel();

// check value is undefined

byteCodeGenerator->Writer()->BrReg2(Js::OpCode::BrSrNeq\_A, skipDefault, valueLocation, funcInfo->undefinedConstantRegister);

// Evaluate the default expression and assign it.

Emit(init, byteCodeGenerator, funcInfo, false);

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, valueLocationTmp, init->location);

funcInfo->ReleaseLoc(init);

// jmp to loadIter

byteCodeGenerator->Writer()->Br(loadIter);

// skipDefault:

byteCodeGenerator->Writer()->MarkLabel(skipDefault);

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, valueLocationTmp, valueLocation);

// loadIter:

// @@iterator

byteCodeGenerator->Writer()->MarkLabel(loadIter);

byteCodeGenerator->EndStatement(init);

if (elem->nop == knopObjectPattern)

{

EmitDestructuredObject(elem, valueLocationTmp, byteCodeGenerator, funcInfo);

}

else

{

// Recursively emit a destructured array using the current .next() as the RHS.

EmitDestructuredArray(elem, valueLocationTmp, byteCodeGenerator, funcInfo);

}

funcInfo->ReleaseTmpRegister(valueLocationTmp);

}

else

{

if (elem->nop == knopObjectPattern)

{

EmitDestructuredObject(elem, valueLocation, byteCodeGenerator, funcInfo);

}

else

{

// Recursively emit a destructured array using the current .next() as the RHS.

EmitDestructuredArray(elem, valueLocation, byteCodeGenerator, funcInfo);

}

}

}

else

{

EmitDestructuredValueOrInitializer(elem, valueLocation, init, byteCodeGenerator, funcInfo);

}

funcInfo->ReleaseTmpRegister(valueLocation);

funcInfo->ReleaseTmpRegister(itemLocation);

byteCodeGenerator->EndStatement(elem);

if (list->nop == knopList)

{

list = list->sxBin.pnode2;

}

else

{

break;

}

}

// If we saw a rest element, emit the rest array.

if (elem != nullptr && elem->nop == knopEllipsis)

{

EmitDestructuredRestArray(elem, iteratorLocation, byteCodeGenerator, funcInfo);

}

funcInfo->ReleaseTmpRegister(iteratorLocation);

byteCodeGenerator->EndStatement(lhs);

}

void EmitNameInvoke(Js::RegSlot lhsLocation,

Js::RegSlot objectLocation,

ParseNodePtr nameNode,

ByteCodeGenerator\* byteCodeGenerator,

FuncInfo\* funcInfo)

{

Assert(nameNode != nullptr);

if (nameNode->nop == knopComputedName)

{

ParseNodePtr pnode1 = nameNode->sxUni.pnode1;

Emit(pnode1, byteCodeGenerator, funcInfo, false/\*isConstructorCall\*/);

byteCodeGenerator->Writer()->Element(Js::OpCode::LdElemI\_A, lhsLocation, objectLocation, pnode1->location);

funcInfo->ReleaseLoc(pnode1);

}

else

{

Assert(nameNode->nop == knopName || nameNode->nop == knopStr);

Symbol \*sym = nameNode->sxPid.sym;

Js::PropertyId propertyId = sym ? sym->EnsurePosition(byteCodeGenerator) : nameNode->sxPid.pid->GetPropertyId();

uint cacheId = funcInfo->FindOrAddInlineCacheId(objectLocation, propertyId, false/\*isLoadMethod\*/, false/\*isStore\*/);

byteCodeGenerator->Writer()->PatchableProperty(Js::OpCode::LdFld, lhsLocation, objectLocation, cacheId);

}

}

void EmitDestructuredValueOrInitializer(ParseNodePtr lhsElementNode,

Js::RegSlot rhsLocation,

ParseNodePtr initializer,

ByteCodeGenerator \*byteCodeGenerator,

FuncInfo \*funcInfo)

{

// If we have initializer we need to see if the destructured value is undefined or not - if it is undefined we need to assign initializer

Js::ByteCodeLabel useDefault = -1;

Js::ByteCodeLabel end = -1;

Js::RegSlot rhsLocationTmp = rhsLocation;

if (initializer != nullptr)

{

rhsLocationTmp = funcInfo->AcquireTmpRegister();

useDefault = byteCodeGenerator->Writer()->DefineLabel();

end = byteCodeGenerator->Writer()->DefineLabel();

byteCodeGenerator->Writer()->BrReg2(Js::OpCode::BrSrEq\_A, useDefault, rhsLocation, funcInfo->undefinedConstantRegister);

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, rhsLocationTmp, rhsLocation);

byteCodeGenerator->Writer()->Br(end);

byteCodeGenerator->Writer()->MarkLabel(useDefault);

Emit(initializer, byteCodeGenerator, funcInfo, false/\*isContructorCall\*/);

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, rhsLocationTmp, initializer->location);

funcInfo->ReleaseLoc(initializer);

byteCodeGenerator->Writer()->MarkLabel(end);

}

if (lhsElementNode->nop == knopArrayPattern)

{

EmitDestructuredArray(lhsElementNode, rhsLocationTmp, byteCodeGenerator, funcInfo);

}

else if (lhsElementNode->nop == knopObjectPattern)

{

EmitDestructuredObject(lhsElementNode, rhsLocationTmp, byteCodeGenerator, funcInfo);

}

else

{

EmitDestructuredElement(lhsElementNode, rhsLocationTmp, byteCodeGenerator, funcInfo);

}

if (initializer != nullptr)

{

funcInfo->ReleaseTmpRegister(rhsLocationTmp);

}

}

void EmitDestructuredObjectMember(ParseNodePtr memberNode,

Js::RegSlot rhsLocation,

ByteCodeGenerator \*byteCodeGenerator,

FuncInfo \*funcInfo)

{

Assert(memberNode->nop == knopObjectPatternMember);

Js::RegSlot nameLocation = funcInfo->AcquireTmpRegister();

EmitNameInvoke(nameLocation, rhsLocation, memberNode->sxBin.pnode1, byteCodeGenerator, funcInfo);

// Imagine we are transforming

// {x:x1} = {} to x1 = {}.x (here x1 is the second node of the member but that is our lhsnode)

ParseNodePtr lhsElementNode = memberNode->sxBin.pnode2;

ParseNodePtr init = nullptr;

if (lhsElementNode->IsVarLetOrConst())

{

init = lhsElementNode->sxVar.pnodeInit;

}

else if (lhsElementNode->nop == knopAsg)

{

init = lhsElementNode->sxBin.pnode2;

lhsElementNode = lhsElementNode->sxBin.pnode1;

}

EmitDestructuredValueOrInitializer(lhsElementNode, nameLocation, init, byteCodeGenerator, funcInfo);

funcInfo->ReleaseTmpRegister(nameLocation);

}

void EmitDestructuredObject(ParseNode \*lhs,

Js::RegSlot rhsLocation,

ByteCodeGenerator \*byteCodeGenerator,

FuncInfo \*funcInfo)

{

Assert(lhs->nop == knopObjectPattern);

ParseNodePtr pnode1 = lhs->sxUni.pnode1;

if (pnode1 != nullptr)

{

Assert(pnode1->nop == knopList || pnode1->nop == knopObjectPatternMember);

byteCodeGenerator->StartStatement(lhs);

ParseNodePtr current = pnode1;

while (current->nop == knopList)

{

ParseNodePtr memberNode = current->sxBin.pnode1;

EmitDestructuredObjectMember(memberNode, rhsLocation, byteCodeGenerator, funcInfo);

current = current->sxBin.pnode2;

}

EmitDestructuredObjectMember(current, rhsLocation, byteCodeGenerator, funcInfo);

byteCodeGenerator->EndStatement(lhs);

}

}

void EmitAssignment(

ParseNode \*asgnNode,

ParseNode \*lhs,

Js::RegSlot rhsLocation,

ByteCodeGenerator \*byteCodeGenerator,

FuncInfo \*funcInfo)

{

switch (lhs->nop)

{

// assignment to a local or global variable

case knopVarDecl:

case knopLetDecl:

case knopConstDecl:

{

Symbol \*sym = lhs->sxVar.sym;

Assert(sym != nullptr);

byteCodeGenerator->EmitPropStore(rhsLocation, sym, nullptr, funcInfo, lhs->nop == knopLetDecl, lhs->nop == knopConstDecl);

break;

}

case knopName:

{

byteCodeGenerator->EmitPropStore(rhsLocation, lhs->sxPid.sym, lhs->sxPid.pid, funcInfo);

break;

}

// x.y =

case knopDot:

{

// PutValue(x, "y", rhs)

Js::PropertyId propertyId = lhs->sxBin.pnode2->sxPid.PropertyIdFromNameNode();

uint cacheId = funcInfo->FindOrAddInlineCacheId(lhs->sxBin.pnode1->location, propertyId, false, true);

if (lhs->sxBin.pnode1->nop == knopSuper)

{

byteCodeGenerator->Writer()->PatchablePropertyWithThisPtr(Js::OpCode::StSuperFld, rhsLocation, lhs->sxBin.pnode1->location, funcInfo->thisPointerRegister, cacheId);

}

else

{

byteCodeGenerator->Writer()->PatchableProperty(

ByteCodeGenerator::GetStFldOpCode(funcInfo, false, false, false, false), rhsLocation, lhs->sxBin.pnode1->location, cacheId);

}

break;

}

case knopIndex:

{

byteCodeGenerator->Writer()->Element(

ByteCodeGenerator::GetStElemIOpCode(funcInfo),

rhsLocation, lhs->sxBin.pnode1->location, lhs->sxBin.pnode2->location);

break;

}

case knopObjectPattern:

{

Assert(byteCodeGenerator->IsES6DestructuringEnabled());

// Copy the rhs value to be the result of the assignment if needed.

if (asgnNode != nullptr)

{

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, asgnNode->location, rhsLocation);

}

return EmitDestructuredObject(lhs, rhsLocation, byteCodeGenerator, funcInfo);

}

case knopArrayPattern:

{

Assert(byteCodeGenerator->IsES6DestructuringEnabled());

// Copy the rhs value to be the result of the assignment if needed.

if (asgnNode != nullptr)

{

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, asgnNode->location, rhsLocation);

}

return EmitDestructuredArray(lhs, rhsLocation, byteCodeGenerator, funcInfo);

}

case knopArray:

case knopObject:

// Assignment to array/object can get through to byte code gen when the parser fails to convert destructuring

// assignment to pattern (because of structural mismatch between LHS & RHS?). Revisit when we nail

// down early vs. runtime errors for destructuring.

byteCodeGenerator->Writer()->W1(Js::OpCode::RuntimeReferenceError, SCODE\_CODE(JSERR\_CantAssignTo));

break;

default:

Assert(!PHASE\_ON1(Js::EarlyReferenceErrorsPhase));

byteCodeGenerator->Writer()->W1(Js::OpCode::RuntimeReferenceError, SCODE\_CODE(JSERR\_CantAssignTo));

break;

}

if (asgnNode != nullptr)

{

// We leave it up to the caller to pass this node only if the assignment expression is used.

if (asgnNode->location != rhsLocation)

{

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, asgnNode->location, rhsLocation);

}

}

}

void EmitLoad(

ParseNode \*lhs,

ByteCodeGenerator \*byteCodeGenerator,

FuncInfo \*funcInfo)

{

// Emit the instructions to load the value into the LHS location. Do not assign/free any temps

// in the process.

// We usually get here as part of an op-equiv expression: x.y += z;

// In such a case, x has to be emitted first, then the value of x.y loaded (by this function), then z emitted.

switch (lhs->nop)

{

// load of a local or global variable

case knopName:

{

funcInfo->AcquireLoc(lhs);

byteCodeGenerator->EmitPropLoad(lhs->location, lhs->sxPid.sym, lhs->sxPid.pid, funcInfo);

break;

}

// = x.y

case knopDot:

{

// get field id for "y"

Js::PropertyId propertyId = lhs->sxBin.pnode2->sxPid.PropertyIdFromNameNode();

funcInfo->AcquireLoc(lhs);

EmitReference(lhs, byteCodeGenerator, funcInfo);

uint cacheId = funcInfo->FindOrAddInlineCacheId(lhs->sxBin.pnode1->location, propertyId, false, false);

byteCodeGenerator->Writer()->PatchableProperty(Js::OpCode::LdFld, lhs->location, lhs->sxBin.pnode1->location, cacheId);

break;

}

case knopIndex:

funcInfo->AcquireLoc(lhs);

EmitReference(lhs, byteCodeGenerator, funcInfo);

byteCodeGenerator->Writer()->Element(

Js::OpCode::LdElemI\_A, lhs->location, lhs->sxBin.pnode1->location, lhs->sxBin.pnode2->location);

break;

// f(x) +=

case knopCall:

funcInfo->AcquireLoc(lhs);

EmitReference(lhs, byteCodeGenerator, funcInfo);

EmitCall(lhs, /\*rhs=\*/ Js::Constants::NoRegister, byteCodeGenerator, funcInfo, /\*fReturnValue=\*/ false, /\*fAssignRegs=\*/ false, /\*fHasNewTarget=\*/ false);

break;

default:

funcInfo->AcquireLoc(lhs);

Emit(lhs, byteCodeGenerator, funcInfo, false);

break;

}

}

void EmitList(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo)

{

if (pnode != nullptr)

{

while (pnode->nop == knopList)

{

byteCodeGenerator->EmitTopLevelStatement(pnode->sxBin.pnode1, funcInfo, false);

pnode = pnode->sxBin.pnode2;

}

byteCodeGenerator->EmitTopLevelStatement(pnode, funcInfo, false);

}

}

void EmitSpreadArgToListBytecodeInstr(ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo, Js::RegSlot argLoc, Js::ProfileId callSiteId, Js::ArgSlot &argIndex)

{

Js::RegSlot regVal = funcInfo->AcquireTmpRegister();

byteCodeGenerator->Writer()->Reg2(Js::OpCode::LdCustomSpreadIteratorList, regVal, argLoc);

byteCodeGenerator->Writer()->ArgOut<true>(++argIndex, regVal, callSiteId);

funcInfo->ReleaseTmpRegister(regVal);

}

size\_t EmitArgs(

ParseNode \*pnode,

BOOL fAssignRegs,

ByteCodeGenerator \*byteCodeGenerator,

FuncInfo \*funcInfo,

Js::ProfileId callSiteId,

Js::AuxArray<uint32> \*spreadIndices = nullptr

)

{

Js::ArgSlot argIndex = 0;

Js::ArgSlot spreadIndex = 0;

if (pnode != nullptr)

{

while (pnode->nop == knopList)

{

// If this is a put, the arguments have already been evaluated (see EmitReference).

// We just need to emit the ArgOut instructions.

if (fAssignRegs)

{

Emit(pnode->sxBin.pnode1, byteCodeGenerator, funcInfo, false);

}

if (pnode->sxBin.pnode1->nop == knopEllipsis)

{

Assert(spreadIndices != nullptr);

spreadIndices->elements[spreadIndex++] = argIndex + 1; // account for 'this'

EmitSpreadArgToListBytecodeInstr(byteCodeGenerator, funcInfo, pnode->sxBin.pnode1->location, callSiteId, argIndex);

}

else

{

byteCodeGenerator->Writer()->ArgOut<true>(++argIndex, pnode->sxBin.pnode1->location, callSiteId);

}

if (fAssignRegs)

{

funcInfo->ReleaseLoc(pnode->sxBin.pnode1);

}

pnode = pnode->sxBin.pnode2;

}

// If this is a put, the call target has already been evaluated (see EmitReference).

if (fAssignRegs)

{

Emit(pnode, byteCodeGenerator, funcInfo, false);

}

if (pnode->nop == knopEllipsis)

{

Assert(spreadIndices != nullptr);

spreadIndices->elements[spreadIndex++] = argIndex + 1; // account for 'this'

EmitSpreadArgToListBytecodeInstr(byteCodeGenerator, funcInfo, pnode->location, callSiteId, argIndex);

}

else

{

byteCodeGenerator->Writer()->ArgOut<true>(++argIndex, pnode->location, callSiteId);

}

if (fAssignRegs)

{

funcInfo->ReleaseLoc(pnode);

}

}

return argIndex;

}

void EmitArgListStart(

Js::RegSlot thisLocation,

ByteCodeGenerator \*byteCodeGenerator,

FuncInfo \*funcInfo,

Js::ProfileId callSiteId)

{

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

{

// Emit the "this" object.

byteCodeGenerator->Writer()->ArgOut<true>(0, thisLocation, callSiteId);

}

}

Js::ArgSlot EmitArgListEnd(

ParseNode \*pnode,

Js::RegSlot rhsLocation,

Js::RegSlot thisLocation,

Js::RegSlot evalLocation,

Js::RegSlot newTargetLocation,

ByteCodeGenerator \*byteCodeGenerator,

FuncInfo \*funcInfo,

size\_t argIndex,

Js::ProfileId callSiteId)

{

BOOL fEvalInModule = false;

BOOL fIsPut = (rhsLocation != Js::Constants::NoRegister);

BOOL fIsEval = (evalLocation != Js::Constants::NoRegister);

BOOL fHasNewTarget = (newTargetLocation != Js::Constants::NoRegister);

Js::ArgSlot argSlotIndex = (Js::ArgSlot) argIndex;

static const Js::ArgSlot maxExtraArgSlot = 4; // max(extraEvalArg, extraArg), where extraEvalArg==2 (moduleRoot,env), extraArg==4 (this, eval, evalInModule, newTarget)

// check for integer overflow with margin for increments below to calculate argument count

if ((size\_t)argSlotIndex != argIndex || argSlotIndex + maxExtraArgSlot < argSlotIndex)

{

Js::Throw::OutOfMemory();

}

Js::ArgSlot evalIndex;

if (fIsPut)

{

// Emit the assigned value as an additional operand. Note that the value has already been evaluated.

// We just need to emit the ArgOut instruction.

argSlotIndex++;

byteCodeGenerator->Writer()->ArgOut<true>(argSlotIndex, rhsLocation, callSiteId);

}

if (fIsEval && argSlotIndex > 0)

{

Assert(!fHasNewTarget);

// Pass the frame display as an extra argument to "eval".

// Do this only if eval is called with some args

Js::RegSlot evalEnv;

if (funcInfo->IsGlobalFunction() && !(funcInfo->GetIsStrictMode() && byteCodeGenerator->GetFlags() & fscrEval))

{

// Use current environment as the environment for the function being called when:

// - this is the root global function (not an eval's global function)

// - this is an eval's global function that is not in strict mode (see else block)

evalEnv = funcInfo->GetEnvRegister();

}

else

{

// Use the frame display as the environment for the function being called when:

// - this is not a global function and thus it will have its own scope

// - this is an eval's global function that is in strict mode, since in strict mode the eval's global function

// has its own scope

evalEnv = funcInfo->frameDisplayRegister;

}

evalEnv = byteCodeGenerator->PrependLocalScopes(evalEnv, evalLocation, funcInfo);

Js::ModuleID moduleID = byteCodeGenerator->GetModuleID();

if (moduleID != kmodGlobal)

{

// Pass both the module root and the environment.

fEvalInModule = true;

byteCodeGenerator->Writer()->ArgOut<true>(argSlotIndex + 1, ByteCodeGenerator::RootObjectRegister, callSiteId);

evalIndex = argSlotIndex + 2;

}

else

{

// Just pass the environment.

evalIndex = argSlotIndex + 1;

}

if (evalEnv == funcInfo->GetEnvRegister() || evalEnv == funcInfo->frameDisplayRegister)

{

byteCodeGenerator->Writer()->ArgOutEnv(evalIndex);

}

else

{

byteCodeGenerator->Writer()->ArgOut<false>(evalIndex, evalEnv, callSiteId);

}

}

if (fHasNewTarget)

{

Assert(!fIsEval);

byteCodeGenerator->Writer()->ArgOut<true>(argSlotIndex + 1, newTargetLocation, callSiteId);

}

Js::ArgSlot argIntCount = argSlotIndex + 1 + (Js::ArgSlot)fIsEval + (Js::ArgSlot)fEvalInModule + (Js::ArgSlot)fHasNewTarget;

// eval and no args passed, return 1 as argument count

if (fIsEval && pnode == nullptr)

{

return 1;

}

return argIntCount;

}

Js::ArgSlot EmitArgList(

ParseNode \*pnode,

Js::RegSlot rhsLocation,

Js::RegSlot thisLocation,

Js::RegSlot newTargetLocation,

BOOL fIsEval,

BOOL fAssignRegs,

ByteCodeGenerator \*byteCodeGenerator,

FuncInfo \*funcInfo,

Js::ProfileId callSiteId,

uint16 spreadArgCount = 0,

Js::AuxArray<uint32> \*\*spreadIndices = nullptr)

{

// This function emits the arguments for a call.

// ArgOut's with uses immediately following defs.

EmitArgListStart(thisLocation, byteCodeGenerator, funcInfo, callSiteId);

Js::RegSlot evalLocation = Js::Constants::NoRegister;

//

// If Emitting arguments for eval and assigning registers, get a tmpLocation for eval.

// This would be used while generating frameDisplay in EmitArgListEnd.

//

if (fIsEval)

{

evalLocation = funcInfo->AcquireTmpRegister();

}

if (spreadArgCount > 0)

{

const size\_t extraAlloc = spreadArgCount \* sizeof(uint32);

Assert(spreadIndices != nullptr);

\*spreadIndices = AnewPlus(byteCodeGenerator->GetAllocator(), extraAlloc, Js::AuxArray<uint32>, spreadArgCount);

}

size\_t argIndex = EmitArgs(pnode, fAssignRegs, byteCodeGenerator, funcInfo, callSiteId, spreadIndices == nullptr ? nullptr : \*spreadIndices);

Js::ArgSlot argumentsCount = EmitArgListEnd(pnode, rhsLocation, thisLocation, evalLocation, newTargetLocation, byteCodeGenerator, funcInfo, argIndex, callSiteId);

if (fIsEval)

{

funcInfo->ReleaseTmpRegister(evalLocation);

}

return argumentsCount;

}

void EmitConstantArgsToVarArray(ByteCodeGenerator \*byteCodeGenerator, \_\_out\_ecount(argCount) Js::Var \*vars, ParseNode \*args, uint argCount)

{

uint index = 0;

while (args->nop == knopList && index < argCount)

{

if (args->sxBin.pnode1->nop == knopInt)

{

int value = args->sxBin.pnode1->sxInt.lw;

vars[index++] = Js::TaggedInt::ToVarUnchecked(value);

}

else if (args->sxBin.pnode1->nop == knopFlt)

{

Js::Var number = Js::JavascriptNumber::New(args->sxBin.pnode1->sxFlt.dbl, byteCodeGenerator->GetScriptContext());

#if ! FLOATVAR

byteCodeGenerator->GetScriptContext()->BindReference(number);

#endif

vars[index++] = number;

}

else

{

AnalysisAssert(false);

}

args = args->sxBin.pnode2;

}

if (index == argCount)

{

Assert(false);

Js::Throw::InternalError();

return;

}

if (args->nop == knopInt)

{

int value = args->sxInt.lw;

vars[index++] = Js::TaggedInt::ToVarUnchecked(value);

}

else if (args->nop == knopFlt)

{

Js::Var number = Js::JavascriptNumber::New(args->sxFlt.dbl, byteCodeGenerator->GetScriptContext());

#if ! FLOATVAR

byteCodeGenerator->GetScriptContext()->BindReference(number);

#endif

vars[index++] = number;

}

else

{

AnalysisAssert(false);

}

}

void EmitConstantArgsToIntArray(ByteCodeGenerator \*byteCodeGenerator, \_\_out\_ecount(argCount) int32 \*vars, ParseNode \*args, uint argCount)

{

uint index = 0;

while (args->nop == knopList && index < argCount)

{

Assert(args->sxBin.pnode1->nop == knopInt);

vars[index++] = args->sxBin.pnode1->sxInt.lw;

args = args->sxBin.pnode2;

}

if (index == argCount)

{

Assert(false);

Js::Throw::InternalError();

return;

}

Assert(args->nop == knopInt);

vars[index++] = args->sxInt.lw;

Assert(index == argCount);

}

void EmitConstantArgsToFltArray(ByteCodeGenerator \*byteCodeGenerator, \_\_out\_ecount(argCount) double \*vars, ParseNode \*args, uint argCount)

{

uint index = 0;

while (args->nop == knopList && index < argCount)

{

OpCode nop = args->sxBin.pnode1->nop;

if (nop == knopInt)

{

vars[index++] = (double)args->sxBin.pnode1->sxInt.lw;

}

else

{

Assert(nop == knopFlt);

vars[index++] = args->sxBin.pnode1->sxFlt.dbl;

}

args = args->sxBin.pnode2;

}

if (index == argCount)

{

Assert(false);

Js::Throw::InternalError();

return;

}

if (args->nop == knopInt)

{

vars[index++] = (double)args->sxInt.lw;

}

else

{

Assert(args->nop == knopFlt);

vars[index++] = args->sxFlt.dbl;

}

Assert(index == argCount);

}

//

// Called when we have new Ctr(constant, constant...)

//

Js::ArgSlot EmitNewObjectOfConstants(

ParseNode \*pnode,

ByteCodeGenerator \*byteCodeGenerator,

FuncInfo \*funcInfo,

unsigned int argCount)

{

EmitArgListStart(Js::Constants::NoRegister, byteCodeGenerator, funcInfo, Js::Constants::NoProfileId);

// Create the vars array

Js::VarArrayVarCount \*vars = AnewPlus(byteCodeGenerator->GetAllocator(), (argCount - 1) \* sizeof(Js::Var), Js::VarArrayVarCount, Js::TaggedInt::ToVarUnchecked(argCount - 1));

// Emit all constants to the vars array

EmitConstantArgsToVarArray(byteCodeGenerator, vars->elements, pnode->sxCall.pnodeArgs, argCount - 1);

// Finish the arg list

Js::ArgSlot actualArgCount = EmitArgListEnd(

pnode->sxCall.pnodeArgs,

Js::Constants::NoRegister,

Js::Constants::NoRegister,

Js::Constants::NoRegister,

Js::Constants::NoRegister,

byteCodeGenerator,

funcInfo,

argCount - 1,

Js::Constants::NoProfileId);

// Make sure the cacheId to regSlot map in the ByteCodeWriter is left in a consistent state after writing NewScObject\_A

byteCodeGenerator->Writer()->RemoveEntryForRegSlotFromCacheIdMap(pnode->sxCall.pnodeTarget->location);

// Generate the opcode with vars

byteCodeGenerator->Writer()->AuxiliaryContext(

Js::OpCode::NewScObject\_A,

funcInfo->AcquireLoc(pnode),

vars,

sizeof(Js::VarArray) + (argCount - 1) \* sizeof(Js::Var),

pnode->sxCall.pnodeTarget->location);

AdeletePlus(byteCodeGenerator->GetAllocator(), (argCount - 1) \* sizeof(Js::VarArrayVarCount), vars);

return actualArgCount;

}

void EmitMethodFld(bool isRoot, bool isScoped, Js::RegSlot location, Js::RegSlot callObjLocation, Js::PropertyId propertyId, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo, bool registerCacheIdForCall = true)

{

Js::OpCode opcode;

if (!isRoot)

{

if (callObjLocation == funcInfo->frameObjRegister)

{

opcode = Js::OpCode::LdLocalMethodFld;

}

else

{

opcode = Js::OpCode::LdMethodFld;

}

}

else if (isScoped)

{

opcode = Js::OpCode::ScopedLdMethodFld;

}

else

{

opcode = Js::OpCode::LdRootMethodFld;

}

if (isScoped || !isRoot)

{

Assert(isScoped || !isRoot || callObjLocation == ByteCodeGenerator::RootObjectRegister);

uint cacheId = funcInfo->FindOrAddInlineCacheId(callObjLocation, propertyId, true, false);

if (callObjLocation == funcInfo->frameObjRegister)

{

byteCodeGenerator->Writer()->ElementP(opcode, location, cacheId, false /\*isCtor\*/, registerCacheIdForCall);

}

else

{

byteCodeGenerator->Writer()->PatchableProperty(opcode, location, callObjLocation, cacheId, false /\*isCtor\*/, registerCacheIdForCall);

}

}

else

{

uint cacheId = funcInfo->FindOrAddRootObjectInlineCacheId(propertyId, true, false);

byteCodeGenerator->Writer()->PatchableRootProperty(opcode, location, cacheId, true, false, registerCacheIdForCall);

}

}

void EmitMethodFld(ParseNode \*pnode, Js::RegSlot callObjLocation, Js::PropertyId propertyId, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo, bool registerCacheIdForCall = true)

{

// Load a call target of the form x.y(). (Call target may be a plain knopName if we're getting it from

// the global object, etc.)

bool isRoot = pnode->nop == knopName && (pnode->sxPid.sym == nullptr || pnode->sxPid.sym->GetIsGlobal());

bool isScoped = (byteCodeGenerator->GetFlags() & fscrEval) != 0 ||

(isRoot && callObjLocation != ByteCodeGenerator::RootObjectRegister);

EmitMethodFld(isRoot, isScoped, pnode->location, callObjLocation, propertyId, byteCodeGenerator, funcInfo, registerCacheIdForCall);

}

// lhs.apply(this, arguments);

void EmitApplyCall(ParseNode\* pnode, Js::RegSlot rhsLocation, ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* funcInfo, BOOL fReturnValue, BOOL fAssignRegs)

{

ParseNode\* applyNode = pnode->sxCall.pnodeTarget;

ParseNode\* thisNode = pnode->sxCall.pnodeArgs->sxBin.pnode1;

Assert(applyNode->nop == knopDot);

ParseNode\* funcNode = applyNode->sxBin.pnode1;

Js::ByteCodeLabel slowPath = byteCodeGenerator->Writer()->DefineLabel();

Js::ByteCodeLabel afterSlowPath = byteCodeGenerator->Writer()->DefineLabel();

Js::ByteCodeLabel argsAlreadyCreated = byteCodeGenerator->Writer()->DefineLabel();

Assert(applyNode->nop == knopDot);

Emit(funcNode, byteCodeGenerator, funcInfo, false);

funcInfo->AcquireLoc(applyNode);

Js::PropertyId propertyId = applyNode->sxBin.pnode2->sxPid.PropertyIdFromNameNode();

// As we won't be emitting a call instruction for apply, no need to register the cacheId for apply

// load to be associated with the call. This is also required, as in the absence of a corresponding

// call for apply, we won't remove the entry for "apply" cacheId from

// ByteCodeWriter::callRegToLdFldCacheIndexMap, which is contrary to our assumption that we would

// have removed an entry from a map upon seeing its corresponding call.

EmitMethodFld(applyNode, funcNode->location, propertyId, byteCodeGenerator, funcInfo, false /\*registerCacheIdForCall\*/);

Symbol \*argSym = funcInfo->GetArgumentsSymbol();

Assert(argSym && argSym->GetIsArguments());

Js::RegSlot argumentsLoc = argSym->GetLocation();

byteCodeGenerator->Writer()->Reg1(Js::OpCode::LdArgumentsFromFrame, argumentsLoc);

byteCodeGenerator->Writer()->BrReg1(Js::OpCode::BrNotNull\_A, argsAlreadyCreated, argumentsLoc);

// If apply is overridden, bail to slow path.

byteCodeGenerator->Writer()->BrReg1(Js::OpCode::BrFncNeqApply, slowPath, applyNode->location);

// Note: acquire and release a temp register for this stack arg pointer instead of trying to stash it

// in funcInfo->stackArgReg. Otherwise, we'll needlessly load and store it in jitted loop bodies and

// may crash if we try to unbox it on the store.

Js::RegSlot stackArgReg = funcInfo->AcquireTmpRegister();

byteCodeGenerator->Writer()->Reg1(Js::OpCode::LdStackArgPtr, stackArgReg);

Js::RegSlot argCountLocation = funcInfo->AcquireTmpRegister();

byteCodeGenerator->Writer()->Reg1(Js::OpCode::LdArgCnt, argCountLocation);

byteCodeGenerator->Writer()->Reg5(Js::OpCode::ApplyArgs, funcNode->location, funcNode->location, thisNode->location, stackArgReg, argCountLocation);

funcInfo->ReleaseTmpRegister(argCountLocation);

funcInfo->ReleaseTmpRegister(stackArgReg);

funcInfo->ReleaseLoc(applyNode);

funcInfo->ReleaseLoc(funcNode);

// Clear these nodes as they are going to be used to re-generate the slow path.

VisitClearTmpRegs(applyNode, byteCodeGenerator, funcInfo);

VisitClearTmpRegs(funcNode, byteCodeGenerator, funcInfo);

byteCodeGenerator->Writer()->Br(afterSlowPath);

// slow path

byteCodeGenerator->Writer()->MarkLabel(slowPath);

if (funcInfo->frameObjRegister != Js::Constants::NoRegister)

{

byteCodeGenerator->EmitScopeObjectInit(funcInfo);

}

byteCodeGenerator->LoadHeapArguments(funcInfo);

byteCodeGenerator->Writer()->MarkLabel(argsAlreadyCreated);

EmitCall(pnode, rhsLocation, byteCodeGenerator, funcInfo, fReturnValue, fAssignRegs,/\*fHasNewTarget\*/false);

byteCodeGenerator->Writer()->MarkLabel(afterSlowPath);

}

void EmitMethodElem(ParseNode \*pnode, Js::RegSlot callObjLocation, Js::RegSlot indexLocation, ByteCodeGenerator \*byteCodeGenerator)

{

// Load a call target of the form x[y]().

byteCodeGenerator->Writer()->Element(Js::OpCode::LdMethodElem, pnode->location, callObjLocation, indexLocation);

}

void EmitCallTargetNoEvalComponents(

ParseNode \*pnodeTarget,

BOOL fSideEffectArgs,

Js::RegSlot \*thisLocation,

Js::RegSlot \*callObjLocation,

ByteCodeGenerator \*byteCodeGenerator,

FuncInfo \*funcInfo)

{

// We first get a reference to the call target, then evaluate the arguments, then

// evaluate the call target.

// - emit reference to target

// - copy instance to scratch reg if necessary.

// - assign this

// - assign instance for dynamic/global name

// - emit args

// - do call (CallFld/Elem/I)

switch (pnodeTarget->nop)

{

case knopDot:

\*thisLocation = pnodeTarget->sxBin.pnode1->location;

\*callObjLocation = pnodeTarget->sxBin.pnode1->location;

break;

case knopIndex:

\*thisLocation = pnodeTarget->sxBin.pnode1->location;

\*callObjLocation = pnodeTarget->sxBin.pnode1->location;

break;

case knopName:

// If the call target is a name, do some extra work to get its instance and the "this" pointer.

byteCodeGenerator->EmitLoadInstance(pnodeTarget->sxPid.sym, pnodeTarget->sxPid.pid, thisLocation, callObjLocation, funcInfo);

if (\*thisLocation == Js::Constants::NoRegister)

{

\*thisLocation = funcInfo->undefinedConstantRegister;

}

if (byteCodeGenerator->GetScriptContext()->GetConfig()->IsLetAndConstEnabled())

{

EmitUseBeforeDeclaration(pnodeTarget, byteCodeGenerator, funcInfo);

}

break;

default:

\*thisLocation = funcInfo->undefinedConstantRegister;

break;

}

}

void EmitSuperMethodBegin(

ParseNode \*pnodeTarget,

ByteCodeGenerator \*byteCodeGenerator,

FuncInfo \*funcInfo)

{

FuncInfo \*parentFuncInfo = funcInfo;

if (parentFuncInfo->IsLambda())

{

parentFuncInfo = byteCodeGenerator->FindEnclosingNonLambda();

}

if (pnodeTarget->sxBin.pnode1->nop == knopSuper && parentFuncInfo->IsClassConstructor() && !parentFuncInfo->IsBaseClassConstructor())

{

byteCodeGenerator->EmitScopeSlotLoadThis(funcInfo, funcInfo->thisPointerRegister, /\*chkUndecl\*/ true);

}

}

void EmitCallTarget(

ParseNode \*pnodeTarget,

BOOL fSideEffectArgs,

Js::RegSlot \*thisLocation,

Js::RegSlot \*callObjLocation,

ByteCodeGenerator \*byteCodeGenerator,

FuncInfo \*funcInfo)

{

// - emit target

// - assign this

// - emit args

// - do call

// The call target is fully evaluated before the argument list. Note that we're not handling

// put-call cases here currently, as such cases only apply to host objects

// and are very unlikely to behave differently depending on the order of evaluation.

switch (pnodeTarget->nop)

{

case knopDot:

{

funcInfo->AcquireLoc(pnodeTarget);

// Assign the call target operand(s), putting them into expression temps if necessary to protect

// them from side-effects.

if (fSideEffectArgs)

{

// Though we're done with target evaluation after this point, still protect opnd1 from

// arg side-effects as it's the "this" pointer.

SaveOpndValue(pnodeTarget->sxBin.pnode1, funcInfo);

}

if ((pnodeTarget->sxBin.pnode2->nop == knopName) && ((pnodeTarget->sxBin.pnode2->sxPid.PropertyIdFromNameNode() == Js::PropertyIds::apply) || (pnodeTarget->sxBin.pnode2->sxPid.PropertyIdFromNameNode() == Js::PropertyIds::call)))

{

pnodeTarget->sxBin.pnode1->SetIsCallApplyTargetLoad();

}

Emit(pnodeTarget->sxBin.pnode1, byteCodeGenerator, funcInfo, false);

Js::PropertyId propertyId = pnodeTarget->sxBin.pnode2->sxPid.PropertyIdFromNameNode();

Js::RegSlot callObjLocation = pnodeTarget->sxBin.pnode1->location;

Js::RegSlot protoLocation = callObjLocation;

EmitSuperMethodBegin(pnodeTarget, byteCodeGenerator, funcInfo);

EmitMethodFld(pnodeTarget, protoLocation, propertyId, byteCodeGenerator, funcInfo);

// Function calls on the 'super' object should maintain current 'this' pointer

\*thisLocation = (pnodeTarget->sxBin.pnode1->nop == knopSuper) ? funcInfo->thisPointerRegister : pnodeTarget->sxBin.pnode1->location;

break;

}

case knopIndex:

{

funcInfo->AcquireLoc(pnodeTarget);

// Assign the call target operand(s), putting them into expression temps if necessary to protect

// them from side-effects.

if (fSideEffectArgs || !(ParseNode::Grfnop(pnodeTarget->sxBin.pnode2->nop) & fnopLeaf))

{

// Though we're done with target evaluation after this point, still protect opnd1 from

// arg or opnd2 side-effects as it's the "this" pointer.

SaveOpndValue(pnodeTarget->sxBin.pnode1, funcInfo);

}

Emit(pnodeTarget->sxBin.pnode1, byteCodeGenerator, funcInfo, false);

Emit(pnodeTarget->sxBin.pnode2, byteCodeGenerator, funcInfo, false);

Js::RegSlot indexLocation = pnodeTarget->sxBin.pnode2->location;

Js::RegSlot callObjLocation = pnodeTarget->sxBin.pnode1->location;

Js::RegSlot protoLocation = callObjLocation;

EmitSuperMethodBegin(pnodeTarget, byteCodeGenerator, funcInfo);

EmitMethodElem(pnodeTarget, protoLocation, indexLocation, byteCodeGenerator);

funcInfo->ReleaseLoc(pnodeTarget->sxBin.pnode2); // don't release indexLocation until after we use it.

// Function calls on the 'super' object should maintain current 'this' pointer

\*thisLocation = (pnodeTarget->sxBin.pnode1->nop == knopSuper) ? funcInfo->thisPointerRegister : pnodeTarget->sxBin.pnode1->location;

break;

}

case knopClassDecl:

{

Emit(pnodeTarget, byteCodeGenerator, funcInfo, false);

// We won't always have an assigned this register (e.g. class expression calls.) We need undefined in this case.

\*thisLocation = funcInfo->thisPointerRegister == Js::Constants::NoRegister ? funcInfo->undefinedConstantRegister : funcInfo->thisPointerRegister;

break;

}

case knopSuper:

{

Emit(pnodeTarget, byteCodeGenerator, funcInfo, false, /\*isConstructorCall\*/ true); // reuse isConstructorCall ("new super()" is illegal)

// Super calls should always use the new.target register unless we don't have one.

// That could happen if we have an eval('super()') outside of a class constructor.

if (funcInfo->newTargetRegister != Js::Constants::NoRegister)

{

\*thisLocation = funcInfo->newTargetRegister;

}

else

{

\*thisLocation = funcInfo->thisPointerRegister;

}

break;

}

case knopName:

{

funcInfo->AcquireLoc(pnodeTarget);

// Assign the call target operand(s), putting them into expression temps if necessary to protect

// them from side-effects.

if (fSideEffectArgs)

{

SaveOpndValue(pnodeTarget, funcInfo);

}

byteCodeGenerator->EmitLoadInstance(pnodeTarget->sxPid.sym, pnodeTarget->sxPid.pid, thisLocation, callObjLocation, funcInfo);

if (\*callObjLocation != Js::Constants::NoRegister)

{

// Load the call target as a property of the instance.

Js::PropertyId propertyId = pnodeTarget->sxPid.PropertyIdFromNameNode();

EmitMethodFld(pnodeTarget, \*callObjLocation, propertyId, byteCodeGenerator, funcInfo);

break;

}

// FALL THROUGH to evaluate call target.

}

default:

// Assign the call target operand(s), putting them into expression temps if necessary to protect

// them from side-effects.

Emit(pnodeTarget, byteCodeGenerator, funcInfo, false);

\*thisLocation = funcInfo->undefinedConstantRegister;

break;

}

// "This" pointer should have been assigned by the above.

Assert(\*thisLocation != Js::Constants::NoRegister);

}

void EmitCallI(

ParseNode \*pnode,

BOOL fEvaluateComponents,

BOOL fIsPut,

BOOL fIsEval,

BOOL fHasNewTarget,

uint32 actualArgCount,

ByteCodeGenerator \*byteCodeGenerator,

FuncInfo \*funcInfo,

Js::ProfileId callSiteId,

Js::AuxArray<uint32> \*spreadIndices = nullptr)

{

// Emit a call where the target is in a register, because it's either a local name or an expression we've

// already evaluated.

ParseNode \*pnodeTarget = pnode->sxBin.pnode1;

Js::OpCode op;

Js::CallFlags callFlags = Js::CallFlags::CallFlags\_None;

uint spreadExtraAlloc = 0;

Js::ArgSlot actualArgSlotCount = (Js::ArgSlot) actualArgCount;

// check for integer overflow

if ((size\_t)actualArgSlotCount != actualArgCount)

{

Js::Throw::OutOfMemory();

}

if (fIsPut)

{

if (pnode->sxCall.spreadArgCount > 0)

{

// TODO(tcare): We are disallowing spread with CallIPut for the moment. See DEVDIV2: 876387

// When CallIPut is migrated to the CallIExtended layout, this can be removed.

byteCodeGenerator->Writer()->W1(Js::OpCode::RuntimeReferenceError, SCODE\_CODE(JSERR\_CantAsgCall));

}

// Grab a tmp register for the call result.

Js::RegSlot tmpReg = funcInfo->AcquireTmpRegister();

byteCodeGenerator->Writer()->CallI(Js::OpCode::CallIFlags, tmpReg, pnodeTarget->location, actualArgSlotCount, callSiteId, Js::CallFlags::CallFlags\_NewTarget);

funcInfo->ReleaseTmpRegister(tmpReg);

}

else

{

if (fEvaluateComponents)

{

// Release the call target operands we assigned above. If we didn't assign them here,

// we'll need them later, so we can't re-use them for the result of the call.

funcInfo->ReleaseLoc(pnodeTarget);

}

// Grab a register for the call result.

if (pnode->isUsed)

{

funcInfo->AcquireLoc(pnode);

}

if (fIsEval)

{

op = Js::OpCode::CallIExtendedFlags;

callFlags = Js::CallFlags::CallFlags\_ExtraArg;

}

else

{

bool isSuperCall = pnodeTarget->nop == knopSuper;

if (isSuperCall)

{

callFlags = Js::CallFlags\_New;

}

if (fHasNewTarget)

{

callFlags = (Js::CallFlags) (callFlags | Js::CallFlags::CallFlags\_ExtraArg | Js::CallFlags::CallFlags\_NewTarget);

}

if (pnode->sxCall.spreadArgCount > 0)

{

op = (isSuperCall || fHasNewTarget) ? Js::OpCode::CallIExtendedFlags : Js::OpCode::CallIExtended;

}

else

{

op = (isSuperCall || fHasNewTarget) ? Js::OpCode::CallIFlags : Js::OpCode::CallI;

}

}

if (op == Js::OpCode::CallI || op == Js::OpCode::CallIFlags)

{

byteCodeGenerator->Writer()->CallI(op, pnode->location, pnodeTarget->location, actualArgSlotCount, callSiteId, callFlags);

}

else

{

uint spreadIndicesSize = 0;

Js::CallIExtendedOptions options = Js::CallIExtended\_None;

if (pnode->sxCall.spreadArgCount > 0)

{

Assert(spreadIndices != nullptr);

spreadExtraAlloc = spreadIndices->count \* sizeof(uint32);

spreadIndicesSize = sizeof(\*spreadIndices) + spreadExtraAlloc;

options = Js::CallIExtended\_SpreadArgs;

}

byteCodeGenerator->Writer()->CallIExtended(op, pnode->location, pnodeTarget->location, actualArgSlotCount, options, spreadIndices, spreadIndicesSize, callSiteId, callFlags);

}

if (pnode->sxCall.spreadArgCount > 0)

{

Assert(spreadExtraAlloc != 0);

AdeletePlus(byteCodeGenerator->GetAllocator(), spreadExtraAlloc, spreadIndices);

}

}

}

void EmitCallInstrNoEvalComponents(

ParseNode \*pnode,

BOOL fIsPut,

BOOL fIsEval,

Js::RegSlot thisLocation,

Js::RegSlot callObjLocation,

uint32 actualArgCount,

ByteCodeGenerator \*byteCodeGenerator,

FuncInfo \*funcInfo,

Js::ProfileId callSiteId,

Js::AuxArray<uint32> \*spreadIndices = nullptr)

{

// Emit the call instruction. The call target is a reference at this point, and we evaluate

// it as part of doing the actual call.

// Note that we don't handle the (fEvaluateComponents == TRUE) case in this function.

// (This function is only called on the !fEvaluateComponents branch in EmitCall.)

ParseNode \*pnodeTarget = pnode->sxBin.pnode1;

switch (pnodeTarget->nop)

{

case knopDot:

{

Assert(pnodeTarget->sxBin.pnode2->nop == knopName);

Js::PropertyId propertyId = pnodeTarget->sxBin.pnode2->sxPid.PropertyIdFromNameNode();

EmitMethodFld(pnodeTarget, callObjLocation, propertyId, byteCodeGenerator, funcInfo);

EmitCallI(pnode, /\*fEvaluateComponents\*/ FALSE, fIsPut, fIsEval, /\*fHasNewTarget\*/ FALSE, actualArgCount, byteCodeGenerator, funcInfo, callSiteId, spreadIndices);

}

break;

case knopIndex:

{

EmitMethodElem(pnodeTarget, pnodeTarget->sxBin.pnode1->location, pnodeTarget->sxBin.pnode2->location, byteCodeGenerator);

EmitCallI(pnode, /\*fEvaluateComponents\*/ FALSE, fIsPut, fIsEval, /\*fHasNewTarget\*/ FALSE, actualArgCount, byteCodeGenerator, funcInfo, callSiteId, spreadIndices);

}

break;

case knopName:

{

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

{

// We still have to get the property from its instance, so emit CallFld.

if (thisLocation != callObjLocation)

{

funcInfo->ReleaseTmpRegister(thisLocation);

}

funcInfo->ReleaseTmpRegister(callObjLocation);

Js::PropertyId propertyId = pnodeTarget->sxPid.PropertyIdFromNameNode();

EmitMethodFld(pnodeTarget, callObjLocation, propertyId, byteCodeGenerator, funcInfo);

EmitCallI(pnode, /\*fEvaluateComponents\*/ FALSE, fIsPut, fIsEval, /\*fHasNewTarget\*/ FALSE, actualArgCount, byteCodeGenerator, funcInfo, callSiteId, spreadIndices);

break;

}

}

// FALL THROUGH

default:

EmitCallI(pnode, /\*fEvaluateComponents\*/ FALSE, fIsPut, fIsEval, /\*fHasNewTarget\*/ FALSE, actualArgCount, byteCodeGenerator, funcInfo, callSiteId, spreadIndices);

break;

}

}

void EmitCallInstr(

ParseNode \*pnode,

BOOL fIsPut,

BOOL fIsEval,

BOOL fHasNewTarget,

Js::RegSlot thisLocation,

Js::RegSlot callObjLocation,

uint32 actualArgCount,

ByteCodeGenerator \*byteCodeGenerator,

FuncInfo \*funcInfo,

Js::ProfileId callSiteId,

Js::AuxArray<uint32> \*spreadIndices = nullptr)

{

// Emit a call instruction. The call target has been fully evaluated already, so we always

// emit a CallI through the register that holds the target value.

// Note that we don't handle !fEvaluateComponents cases at this point.

// (This function is only called on the fEvaluateComponents branch in EmitCall.)

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

{

funcInfo->ReleaseTmpRegister(thisLocation);

}

if (callObjLocation != Js::Constants::NoRegister &&

callObjLocation != thisLocation)

{

funcInfo->ReleaseTmpRegister(callObjLocation);

}

EmitCallI(pnode, /\*fEvaluateComponents\*/ TRUE, fIsPut, fIsEval, fHasNewTarget, actualArgCount, byteCodeGenerator, funcInfo, callSiteId, spreadIndices);

}

void EmitCall(

ParseNode\* pnode,

Js::RegSlot rhsLocation,

ByteCodeGenerator\* byteCodeGenerator,

FuncInfo\* funcInfo,

BOOL fReturnValue,

BOOL fEvaluateComponents,

BOOL fHasNewTarget,

Js::RegSlot overrideThisLocation)

{

BOOL fIsPut = (rhsLocation != Js::Constants::NoRegister);

// If the call returns a float, we'll note this in the byte code.

Js::RegSlot thisLocation = Js::Constants::NoRegister;

Js::RegSlot callObjLocation = Js::Constants::NoRegister;

Js::RegSlot newTargetLocation = Js::Constants::NoRegister;

BOOL fSideEffectArgs = FALSE;

ParseNode \*pnodeTarget = pnode->sxCall.pnodeTarget;

ParseNode \*pnodeArgs = pnode->sxCall.pnodeArgs;

uint16 spreadArgCount = pnode->sxCall.spreadArgCount;

unsigned int argCount = CountArguments(pnode->sxCall.pnodeArgs, &fSideEffectArgs) + (unsigned int)fIsPut;

BOOL fIsEval = !fIsPut && pnode->sxCall.isEvalCall;

if (fIsEval)

{

Assert(!fHasNewTarget);

//

// "eval" takes the closure environment as an extra argument

// Pass the closure env only if some argument is passed

// For just eval(), don't pass the closure environment

//

if (argCount > 1)

{

// Check the module ID as well. If it's not the global (default) module,

// we need to pass the root to eval so it can do the right global lookups.

// (Passing the module root is the least disruptive way to get the module ID

// to the helper, given the current set of byte codes. Once we have a full set

// of byte code ops taking immediate opnds, passing the ID is more intuitive.)

Js::ModuleID moduleID = byteCodeGenerator->GetModuleID();

if (moduleID == kmodGlobal)

{

argCount++;

}

else

{

// Module ID must be passed

argCount += 2;

}

}

}

if (fHasNewTarget)

{

Assert(!fIsEval);

// When we need to pass new.target explicitly, it is passed as an extra argument.

// This is similar to how eval passes an extra argument for the frame display and is

// used to support cases where we need to pass both 'this' and new.target as part of

// a function call.

// OpCode::LdNewTarget knows how to look at the call flags and fetch this argument.

argCount++;

newTargetLocation = funcInfo->newTargetRegister;

Assert(newTargetLocation != Js::Constants::NoRegister);

}

Js::ArgSlot argSlotCount = (Js::ArgSlot)argCount;

if (argCount != (unsigned int)argSlotCount)

{

Js::Throw::OutOfMemory();

}

if (fReturnValue)

{

pnode->isUsed = true;

}

//

// Set up the call.

//

if (!fEvaluateComponents)

{

EmitCallTargetNoEvalComponents(pnodeTarget, fSideEffectArgs, &thisLocation, &callObjLocation, byteCodeGenerator, funcInfo);

}

else

{

EmitCallTarget(pnodeTarget, fSideEffectArgs, &thisLocation, &callObjLocation, byteCodeGenerator, funcInfo);

}

bool releaseThisLocation = true;

// If we are strictly overriding the this location, ignore what the call target set this location to.

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

{

thisLocation = overrideThisLocation;

releaseThisLocation = false;

}

// Evaluate the arguments (nothing mode-specific here).

// Start call, allocate out param space

funcInfo->StartRecordingOutArgs(argSlotCount);

Js::ProfileId callSiteId = byteCodeGenerator->GetNextCallSiteId(Js::OpCode::CallI);

byteCodeGenerator->Writer()->StartCall(Js::OpCode::StartCall, argSlotCount);

Js::AuxArray<uint32> \*spreadIndices;

Js::ArgSlot actualArgCount = EmitArgList(pnodeArgs, rhsLocation, thisLocation, newTargetLocation, fIsEval, fEvaluateComponents, byteCodeGenerator, funcInfo, callSiteId, spreadArgCount, &spreadIndices);

Assert(argSlotCount == actualArgCount);

if (!fEvaluateComponents)

{

EmitCallInstrNoEvalComponents(pnode, fIsPut, fIsEval, thisLocation, callObjLocation, actualArgCount, byteCodeGenerator, funcInfo, callSiteId, spreadIndices);

}

else

{

EmitCallInstr(pnode, fIsPut, fIsEval, fHasNewTarget, releaseThisLocation ? thisLocation : Js::Constants::NoRegister, callObjLocation, actualArgCount, byteCodeGenerator, funcInfo, callSiteId, spreadIndices);

}

// End call, pop param space

funcInfo->EndRecordingOutArgs(argSlotCount);

}

void EmitInvoke(

Js::RegSlot location,

Js::RegSlot callObjLocation,

Js::PropertyId propertyId,

ByteCodeGenerator\* byteCodeGenerator,

FuncInfo\* funcInfo)

{

EmitMethodFld(false, false, location, callObjLocation, propertyId, byteCodeGenerator, funcInfo);

funcInfo->StartRecordingOutArgs(1);

Js::ProfileId callSiteId = byteCodeGenerator->GetNextCallSiteId(Js::OpCode::CallI);

byteCodeGenerator->Writer()->StartCall(Js::OpCode::StartCall, 1);

EmitArgListStart(callObjLocation, byteCodeGenerator, funcInfo, callSiteId);

byteCodeGenerator->Writer()->CallI(Js::OpCode::CallI, location, location, 1, callSiteId);

}

void EmitInvoke(

Js::RegSlot location,

Js::RegSlot callObjLocation,

Js::PropertyId propertyId,

ByteCodeGenerator\* byteCodeGenerator,

FuncInfo\* funcInfo,

Js::RegSlot arg1Location)

{

EmitMethodFld(false, false, location, callObjLocation, propertyId, byteCodeGenerator, funcInfo);

funcInfo->StartRecordingOutArgs(2);

Js::ProfileId callSiteId = byteCodeGenerator->GetNextCallSiteId(Js::OpCode::CallI);

byteCodeGenerator->Writer()->StartCall(Js::OpCode::StartCall, 2);

EmitArgListStart(callObjLocation, byteCodeGenerator, funcInfo, callSiteId);

byteCodeGenerator->Writer()->ArgOut<true>(1, arg1Location, callSiteId);

byteCodeGenerator->Writer()->CallI(Js::OpCode::CallI, location, location, 2, callSiteId);

}

void EmitComputedFunctionNameVar(ParseNode \*nameNode, ParseNode \*exprNode, ByteCodeGenerator \*byteCodeGenerator)

{

AssertMsg(exprNode != nullptr, "callers of this function should pass in a valid expression Node");

if (nameNode == nullptr)

{

return;

}

if ((exprNode->nop == knopFncDecl && (exprNode->sxFnc.pnodeName == nullptr || exprNode->sxFnc.pnodeName->nop != knopVarDecl)))

{

byteCodeGenerator->Writer()->Reg2(Js::OpCode::SetComputedNameVar, exprNode->location, nameNode->location);

}

}

void EmitMemberNode(ParseNode \*memberNode, Js::RegSlot objectLocation, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo, ParseNode\* parentNode, bool useStore, bool\* isObjectEmpty = nullptr)

{

ParseNode \*nameNode = memberNode->sxBin.pnode1;

ParseNode \*exprNode = memberNode->sxBin.pnode2;

bool isFncDecl = exprNode->nop == knopFncDecl;

bool isClassMember = isFncDecl && exprNode->sxFnc.IsClassMember();

// Moved SetComputedNameVar before LdFld of prototype because loading the prototype undefers the function TypeHandler

// which makes this bytecode too late to influence the function.name.

if (nameNode->nop == knopComputedName)

{

// Computed property name

// Transparently pass the name expr

// The Emit will replace this with a temp register if necessary to preserve the value.

nameNode->location = nameNode->sxUni.pnode1->location;

EmitBinaryOpnds(nameNode, exprNode, byteCodeGenerator, funcInfo);

if (isFncDecl && !exprNode->sxFnc.IsClassConstructor())

{

EmitComputedFunctionNameVar(nameNode, exprNode, byteCodeGenerator);

}

}

// Classes allocates a RegSlot as part of Instance Methods EmitClassInitializers,

// but if we don't have any members then we don't need to load the prototype.

Assert(isClassMember == (isObjectEmpty != nullptr));

if (isClassMember && \*isObjectEmpty)

{

\*isObjectEmpty = false;

int cacheId = funcInfo->FindOrAddInlineCacheId(parentNode->location, Js::PropertyIds::prototype, false, false);

byteCodeGenerator->Writer()->PatchableProperty(Js::OpCode::LdFld, objectLocation, parentNode->location, cacheId);

}

if (nameNode->nop == knopComputedName)

{

Assert(memberNode->nop == knopGetMember || memberNode->nop == knopSetMember || memberNode->nop == knopMember);

Js::OpCode setOp = memberNode->nop == knopGetMember ?

(isClassMember ? Js::OpCode::InitClassMemberGetComputedName : Js::OpCode::InitGetElemI) :

memberNode->nop == knopSetMember ?

(isClassMember ? Js::OpCode::InitClassMemberSetComputedName : Js::OpCode::InitSetElemI) :

(isClassMember ? Js::OpCode::InitClassMemberComputedName : Js::OpCode::InitComputedProperty);

byteCodeGenerator->Writer()->Element(setOp, exprNode->location, objectLocation, nameNode->location, true);

// Class and object members need a reference back to the class.

if (isFncDecl)

{

byteCodeGenerator->Writer()->Reg2(Js::OpCode::SetHomeObj, exprNode->location, objectLocation);

}

funcInfo->ReleaseLoc(exprNode);

funcInfo->ReleaseLoc(nameNode);

return;

}

Js::OpCode stFldOpCode = (Js::OpCode)0;

if (useStore)

{

stFldOpCode = ByteCodeGenerator::GetStFldOpCode(funcInfo, false, false, false, isClassMember);

}

Emit(exprNode, byteCodeGenerator, funcInfo, false);

Js::PropertyId propertyId = nameNode->sxPid.PropertyIdFromNameNode();

if (Js::PropertyIds::name == propertyId

&& exprNode->nop == knopFncDecl

&& exprNode->sxFnc.IsStaticMember()

&& parentNode != nullptr && parentNode->nop == knopClassDecl

&& parentNode->sxClass.pnodeConstructor != nullptr)

{

Js::ParseableFunctionInfo\* nameFunc = parentNode->sxClass.pnodeConstructor->sxFnc.funcInfo->byteCodeFunction->GetParseableFunctionInfo();

nameFunc->SetIsStaticNameFunction(true);

}

if (memberNode->nop == knopMember || memberNode->nop == knopMemberShort)

{

// The internal prototype should be set only if the production is of the form PropertyDefinition : PropertyName : AssignmentExpression

if (propertyId == Js::PropertyIds::\_\_proto\_\_ && memberNode->nop != knopMemberShort && (exprNode->nop != knopFncDecl || !exprNode->sxFnc.IsMethod()))

{

byteCodeGenerator->Writer()->Property(Js::OpCode::InitProto, exprNode->location, objectLocation,

funcInfo->FindOrAddReferencedPropertyId(propertyId));

}

else

{

uint cacheId = funcInfo->FindOrAddInlineCacheId(objectLocation, propertyId, false, true);

Js::OpCode patchablePropertyOpCode;

if (useStore)

{

patchablePropertyOpCode = stFldOpCode;

}

else if (isClassMember)

{

patchablePropertyOpCode = Js::OpCode::InitClassMember;

}

else

{

patchablePropertyOpCode = Js::OpCode::InitFld;

}

byteCodeGenerator->Writer()->PatchableProperty(patchablePropertyOpCode, exprNode->location, objectLocation, cacheId);

}

}

else

{

Assert(memberNode->nop == knopGetMember || memberNode->nop == knopSetMember);

Js::OpCode setOp = memberNode->nop == knopGetMember ?

(isClassMember ? Js::OpCode::InitClassMemberGet : Js::OpCode::InitGetFld) :

(isClassMember ? Js::OpCode::InitClassMemberSet : Js::OpCode::InitSetFld);

byteCodeGenerator->Writer()->Property(setOp, exprNode->location, objectLocation, funcInfo->FindOrAddReferencedPropertyId(propertyId));

}

// Class and object members need a reference back to the class.

if (isFncDecl)

{

byteCodeGenerator->Writer()->Reg2(Js::OpCode::SetHomeObj, exprNode->location, objectLocation);

}

funcInfo->ReleaseLoc(exprNode);

if (propertyId == Js::PropertyIds::valueOf)

{

byteCodeGenerator->GetScriptContext()->optimizationOverrides.SetSideEffects(Js::SideEffects\_ValueOf);

}

else if (propertyId == Js::PropertyIds::toString)

{

byteCodeGenerator->GetScriptContext()->optimizationOverrides.SetSideEffects(Js::SideEffects\_ToString);

}

}

void EmitClassInitializers(ParseNode \*memberList, Js::RegSlot objectLocation, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo, ParseNode\* parentNode, bool isObjectEmpty)

{

if (memberList != nullptr)

{

while (memberList->nop == knopList)

{

ParseNode \*memberNode = memberList->sxBin.pnode1;

EmitMemberNode(memberNode, objectLocation, byteCodeGenerator, funcInfo, parentNode, /\*useStore\*/ false, &isObjectEmpty);

memberList = memberList->sxBin.pnode2;

}

EmitMemberNode(memberList, objectLocation, byteCodeGenerator, funcInfo, parentNode, /\*useStore\*/ false, &isObjectEmpty);

}

}

void EmitObjectInitializers(ParseNode \*memberList, Js::RegSlot objectLocation, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo)

{

ParseNode \*pmemberList = memberList;

unsigned int argCount = 0;

uint32 value;

Js::PropertyId propertyId;

//

// 1. Add all non-int property ids to a dictionary propertyIds with value true

// 2. Get the count of propertyIds

// 3. Create a propertyId array of size count

// 4. Put the propIds in the auxiliary area

// 5. Get the objectLiteralCacheId

// 6. Generate propId inits with values

//

// Handle propertyId collision

typedef JsUtil::BaseHashSet<Js::PropertyId, ArenaAllocator, PowerOf2SizePolicy> PropertyIdSet;

PropertyIdSet\* propertyIds = Anew(byteCodeGenerator->GetAllocator(), PropertyIdSet, byteCodeGenerator->GetAllocator(), 17);

bool hasComputedName = false;

if (memberList != nullptr)

{

while (memberList->nop == knopList)

{

if (memberList->sxBin.pnode1->sxBin.pnode1->nop == knopComputedName)

{

hasComputedName = true;

break;

}

propertyId = memberList->sxBin.pnode1->sxBin.pnode1->sxPid.PropertyIdFromNameNode();

if (!byteCodeGenerator->GetScriptContext()->IsNumericPropertyId(propertyId, &value))

{

propertyIds->Item(propertyId);

}

memberList = memberList->sxBin.pnode2;

}

if (memberList->sxBin.pnode1->nop != knopComputedName && !hasComputedName)

{

propertyId = memberList->sxBin.pnode1->sxPid.PropertyIdFromNameNode();

if (!byteCodeGenerator->GetScriptContext()->IsNumericPropertyId(propertyId, &value))

{

propertyIds->Item(propertyId);

}

}

}

argCount = propertyIds->Count();

memberList = pmemberList;

if ((memberList == nullptr) || (argCount == 0))

{

// Empty literal or numeric property only object literal

byteCodeGenerator->Writer()->Reg1(Js::OpCode::NewScObjectSimple, objectLocation);

}

else

{

Js::PropertyIdArray \*propIds = AnewPlus(byteCodeGenerator->GetAllocator(), argCount \* sizeof(Js::PropertyId), Js::PropertyIdArray, argCount);

if (propertyIds->ContainsKey(Js::PropertyIds::\_\_proto\_\_))

{

// Always record whether the initializer contains \_\_proto\_\_ no matter if current environment has it enabled

// or not, in case the bytecode is later run with \_\_proto\_\_ enabled.

propIds->has\_\_proto\_\_ = true;

}

unsigned int argIndex = 0;

while (memberList->nop == knopList)

{

if (memberList->sxBin.pnode1->sxBin.pnode1->nop == knopComputedName)

{

break;

}

propertyId = memberList->sxBin.pnode1->sxBin.pnode1->sxPid.PropertyIdFromNameNode();

if (!byteCodeGenerator->GetScriptContext()->IsNumericPropertyId(propertyId, &value) && propertyIds->Remove(propertyId))

{

propIds->elements[argIndex] = propertyId;

argIndex++;

}

memberList = memberList->sxBin.pnode2;

}

if (memberList->sxBin.pnode1->nop != knopComputedName && !hasComputedName)

{

propertyId = memberList->sxBin.pnode1->sxPid.PropertyIdFromNameNode();

if (!byteCodeGenerator->GetScriptContext()->IsNumericPropertyId(propertyId, &value) && propertyIds->Remove(propertyId))

{

propIds->elements[argIndex] = propertyId;

argIndex++;

}

}

uint32 literalObjectId = funcInfo->GetParsedFunctionBody()->NewObjectLiteral();

// Generate the opcode with propIds and cacheId

byteCodeGenerator->Writer()->Auxiliary(Js::OpCode::NewScObjectLiteral, objectLocation, propIds, sizeof(Js::PropertyIdArray) + argCount \* sizeof(Js::PropertyId), literalObjectId);

Adelete(byteCodeGenerator->GetAllocator(), propertyIds);

AdeletePlus(byteCodeGenerator->GetAllocator(), argCount \* sizeof(Js::PropertyId), propIds);

}

memberList = pmemberList;

bool useStore = false;

// Generate the actual assignment to those properties

if (memberList != nullptr)

{

while (memberList->nop == knopList)

{

ParseNode \*memberNode = memberList->sxBin.pnode1;

if (memberNode->sxBin.pnode1->nop == knopComputedName)

{

useStore = true;

}

byteCodeGenerator->StartSubexpression(memberNode);

EmitMemberNode(memberNode, objectLocation, byteCodeGenerator, funcInfo, nullptr, useStore);

byteCodeGenerator->EndSubexpression(memberNode);

memberList = memberList->sxBin.pnode2;

}

byteCodeGenerator->StartSubexpression(memberList);

EmitMemberNode(memberList, objectLocation, byteCodeGenerator, funcInfo, nullptr, useStore);

byteCodeGenerator->EndSubexpression(memberList);

}

}

void EmitStringTemplate(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo)

{

Assert(pnode->sxStrTemplate.pnodeStringLiterals);

// For a tagged string template, we will create the callsite constant object as part of the FunctionBody constants table.

// We only need to emit code for non-tagged string templates here.

if (!pnode->sxStrTemplate.isTaggedTemplate)

{

// If we have no substitutions and this is not a tagged template, we can emit just the single cooked string.

if (pnode->sxStrTemplate.pnodeSubstitutionExpressions == nullptr)

{

Assert(pnode->sxStrTemplate.pnodeStringLiterals->nop != knopList);

funcInfo->AcquireLoc(pnode);

Emit(pnode->sxStrTemplate.pnodeStringLiterals, byteCodeGenerator, funcInfo, false);

Assert(pnode->location != pnode->sxStrTemplate.pnodeStringLiterals->location);

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, pnode->location, pnode->sxStrTemplate.pnodeStringLiterals->location);

funcInfo->ReleaseLoc(pnode->sxStrTemplate.pnodeStringLiterals);

}

else

{

// If we have substitutions but no tag function, we can skip the callSite object construction (and also ignore raw strings).

funcInfo->AcquireLoc(pnode);

// First string must be a list node since we have substitutions.

AssertMsg(pnode->sxStrTemplate.pnodeStringLiterals->nop == knopList, "First string in the list must be a knopList node.");

ParseNode\* stringNodeList = pnode->sxStrTemplate.pnodeStringLiterals;

// Emit the first string and load that into the pnode location.

Emit(stringNodeList->sxBin.pnode1, byteCodeGenerator, funcInfo, false);

Assert(pnode->location != stringNodeList->sxBin.pnode1->location);

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, pnode->location, stringNodeList->sxBin.pnode1->location);

funcInfo->ReleaseLoc(stringNodeList->sxBin.pnode1);

ParseNode\* expressionNodeList = pnode->sxStrTemplate.pnodeSubstitutionExpressions;

ParseNode\* stringNode;

ParseNode\* expressionNode;

// Now append the substitution expressions and remaining string constants via normal add operator

// We will always have one more string constant than substitution expression

// `strcon1 ${expr1} strcon2 ${expr2} strcon3` = strcon1 + expr1 + strcon2 + expr2 + strcon3

//

// strcon1 --- step 1 (above)

// expr1 \\_\_ step 2

// strcon2 /

// expr2 \\_\_ step 3

// strcon3 /

while (stringNodeList->nop == knopList)

{

// If the current head of the expression list is a list, fetch the node and walk the list.

if (expressionNodeList->nop == knopList)

{

expressionNode = expressionNodeList->sxBin.pnode1;

expressionNodeList = expressionNodeList->sxBin.pnode2;

}

else

{

// This is the last element of the expression list.

expressionNode = expressionNodeList;

}

// Emit the expression and append it to the string we're building.

Emit(expressionNode, byteCodeGenerator, funcInfo, false);

Js::RegSlot toStringLocation = funcInfo->AcquireTmpRegister();

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Conv\_Str, toStringLocation, expressionNode->location);

byteCodeGenerator->Writer()->Reg3(Js::OpCode::Add\_A, pnode->location, pnode->location, toStringLocation);

funcInfo->ReleaseTmpRegister(toStringLocation);

funcInfo->ReleaseLoc(expressionNode);

// Move to the next string in the list - we already got ahead of the expressions in the first string literal above.

stringNodeList = stringNodeList->sxBin.pnode2;

// If the current head of the string literal list is also a list node, need to fetch the actual string literal node.

if (stringNodeList->nop == knopList)

{

stringNode = stringNodeList->sxBin.pnode1;

}

else

{

// This is the last element of the string literal list.

stringNode = stringNodeList;

}

// Emit the string node following the previous expression and append it to the string.

// This is either just some string in the list or it is the last string.

Emit(stringNode, byteCodeGenerator, funcInfo, false);

byteCodeGenerator->Writer()->Reg3(Js::OpCode::Add\_A, pnode->location, pnode->location, stringNode->location);

funcInfo->ReleaseLoc(stringNode);

}

}

}

}

void SetNewArrayElements(ParseNode \*pnode, Js::RegSlot arrayLocation, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo)

{

ParseNode \*args = pnode->sxUni.pnode1;

uint argCount = pnode->sxArrLit.count;

uint spreadCount = pnode->sxArrLit.spreadCount;

bool nativeArrays = CreateNativeArrays(byteCodeGenerator, funcInfo);

bool arrayIntOpt = nativeArrays && pnode->sxArrLit.arrayOfInts;

if (arrayIntOpt)

{

int extraAlloc = argCount \* sizeof(int32);

Js::AuxArray<int> \*ints = AnewPlus(byteCodeGenerator->GetAllocator(), extraAlloc, Js::AuxArray<int32>, argCount);

EmitConstantArgsToIntArray(byteCodeGenerator, ints->elements, args, argCount);

Assert(!pnode->sxArrLit.hasMissingValues);

byteCodeGenerator->Writer()->Auxiliary(

Js::OpCode::NewScIntArray,

pnode->location,

ints,

sizeof(Js::AuxArray<int>) + extraAlloc,

argCount);

AdeletePlus(byteCodeGenerator->GetAllocator(), extraAlloc, ints);

return;

}

bool arrayNumOpt = nativeArrays && pnode->sxArrLit.arrayOfNumbers;

if (arrayNumOpt)

{

int extraAlloc = argCount \* sizeof(double);

Js::AuxArray<double> \*doubles = AnewPlus(byteCodeGenerator->GetAllocator(), extraAlloc, Js::AuxArray<double>, argCount);

EmitConstantArgsToFltArray(byteCodeGenerator, doubles->elements, args, argCount);

Assert(!pnode->sxArrLit.hasMissingValues);

byteCodeGenerator->Writer()->Auxiliary(

Js::OpCode::NewScFltArray,

pnode->location,

doubles,

sizeof(Js::AuxArray<double>) + extraAlloc,

argCount);

AdeletePlus(byteCodeGenerator->GetAllocator(), extraAlloc, doubles);

return;

}

bool arrayLitOpt = pnode->sxArrLit.arrayOfTaggedInts && pnode->sxArrLit.count > 1;

Assert(!arrayLitOpt || !nativeArrays);

Js::RegSlot spreadArrLoc = arrayLocation;

Js::AuxArray<uint32> \*spreadIndices = nullptr;

const uint extraAlloc = spreadCount \* sizeof(uint32);

if (pnode->sxArrLit.spreadCount > 0)

{

arrayLocation = funcInfo->AcquireTmpRegister();

spreadIndices = AnewPlus(byteCodeGenerator->GetAllocator(), extraAlloc, Js::AuxArray<uint32>, spreadCount);

}

byteCodeGenerator->Writer()->Reg1Unsigned1(

pnode->sxArrLit.hasMissingValues ? Js::OpCode::NewScArrayWithMissingValues : Js::OpCode::NewScArray,

arrayLocation,

argCount);

if (args != nullptr)

{

Js::OpCode opcode;

Js::RegSlot arrLoc;

if (argCount == 1 && !byteCodeGenerator->Writer()->DoProfileNewScArrayOp(Js::OpCode::NewScArray))

{

opcode = Js::OpCode::StArrItemC\_CI4;

arrLoc = arrayLocation;

}

else if (arrayLitOpt)

{

opcode = Js::OpCode::StArrSegItem\_A;

arrLoc = funcInfo->AcquireTmpRegister();

byteCodeGenerator->Writer()->Reg2(Js::OpCode::LdArrHead, arrLoc, arrayLocation);

}

else if (Js::JavascriptArray::HasInlineHeadSegment(argCount))

{

// The head segment will be allocated inline as an interior pointer. To keep the array alive, the set operation

// should be done relative to the array header to keep it alive (instead of the array segment).

opcode = Js::OpCode::StArrInlineItem\_CI4;

arrLoc = arrayLocation;

}

else if (argCount <= Js::JavascriptArray::MaxInitialDenseLength)

{

opcode = Js::OpCode::StArrSegItem\_CI4;

arrLoc = funcInfo->AcquireTmpRegister();

byteCodeGenerator->Writer()->Reg2(Js::OpCode::LdArrHead, arrLoc, arrayLocation);

}

else

{

opcode = Js::OpCode::StArrItemI\_CI4;

arrLoc = arrayLocation;

}

if (arrayLitOpt)

{

Js::VarArray \*vars = AnewPlus(byteCodeGenerator->GetAllocator(), argCount \* sizeof(Js::Var), Js::VarArray, argCount);

EmitConstantArgsToVarArray(byteCodeGenerator, vars->elements, args, argCount);

// Generate the opcode with vars

byteCodeGenerator->Writer()->Auxiliary(Js::OpCode::StArrSegItem\_A, arrLoc, vars, sizeof(Js::VarArray) + argCount \* sizeof(Js::Var), argCount);

AdeletePlus(byteCodeGenerator->GetAllocator(), argCount \* sizeof(Js::Var), vars);

}

else

{

uint i = 0;

unsigned spreadIndex = 0;

Js::RegSlot rhsLocation;

while (args->nop == knopList)

{

if (args->sxBin.pnode1->nop != knopEmpty)

{

Emit(args->sxBin.pnode1, byteCodeGenerator, funcInfo, false);

rhsLocation = args->sxBin.pnode1->location;

Js::RegSlot regVal = rhsLocation;

if (args->sxBin.pnode1->nop == knopEllipsis)

{

AnalysisAssert(spreadIndices);

regVal = funcInfo->AcquireTmpRegister();

byteCodeGenerator->Writer()->Reg2(Js::OpCode::LdCustomSpreadIteratorList, regVal, rhsLocation);

spreadIndices->elements[spreadIndex++] = i;

}

byteCodeGenerator->Writer()->ElementUnsigned1(opcode, regVal, arrLoc, i);

if (args->sxBin.pnode1->nop == knopEllipsis)

{

funcInfo->ReleaseTmpRegister(regVal);

}

funcInfo->ReleaseLoc(args->sxBin.pnode1);

}

args = args->sxBin.pnode2;

i++;

}

if (args->nop != knopEmpty)

{

Emit(args, byteCodeGenerator, funcInfo, false);

rhsLocation = args->location;

Js::RegSlot regVal = rhsLocation;

if (args->nop == knopEllipsis)

{

regVal = funcInfo->AcquireTmpRegister();

byteCodeGenerator->Writer()->Reg2(Js::OpCode::LdCustomSpreadIteratorList, regVal, rhsLocation);

AnalysisAssert(spreadIndices);

spreadIndices->elements[spreadIndex] = i;

}

byteCodeGenerator->Writer()->ElementUnsigned1(opcode, regVal, arrLoc, i);

if (args->nop == knopEllipsis)

{

funcInfo->ReleaseTmpRegister(regVal);

}

funcInfo->ReleaseLoc(args);

i++;

}

Assert(i <= argCount);

}

if (arrLoc != arrayLocation)

{

funcInfo->ReleaseTmpRegister(arrLoc);

}

}

if (pnode->sxArrLit.spreadCount > 0)

{

byteCodeGenerator->Writer()->Reg2Aux(Js::OpCode::SpreadArrayLiteral, spreadArrLoc, arrayLocation, spreadIndices, sizeof(Js::AuxArray<uint32>) + extraAlloc, extraAlloc);

AdeletePlus(byteCodeGenerator->GetAllocator(), extraAlloc, spreadIndices);

funcInfo->ReleaseTmpRegister(arrayLocation);

}

}

// FIX: TODO: mixed-mode expressions (arithmetic expressions mixed with boolean expressions); current solution

// will not short-circuit in some cases and is not complete (for example: var i=(x==y))

// This uses Aho and Ullman style double-branch generation (p. 494 ASU); we will need to peephole optimize or replace

// with special case for single-branch style.

void EmitBooleanExpression(ParseNode \*expr, Js::ByteCodeLabel trueLabel, Js::ByteCodeLabel falseLabel, ByteCodeGenerator \*byteCodeGenerator,

FuncInfo \*funcInfo)

{

switch (expr->nop)

{

case knopLogOr:

{

byteCodeGenerator->StartStatement(expr);

Js::ByteCodeLabel leftFalse = byteCodeGenerator->Writer()->DefineLabel();

EmitBooleanExpression(expr->sxBin.pnode1, trueLabel, leftFalse, byteCodeGenerator, funcInfo);

funcInfo->ReleaseLoc(expr->sxBin.pnode1);

byteCodeGenerator->Writer()->MarkLabel(leftFalse);

EmitBooleanExpression(expr->sxBin.pnode2, trueLabel, falseLabel, byteCodeGenerator, funcInfo);

funcInfo->ReleaseLoc(expr->sxBin.pnode2);

byteCodeGenerator->EndStatement(expr);

break;

}

case knopLogAnd:

{

byteCodeGenerator->StartStatement(expr);

Js::ByteCodeLabel leftTrue = byteCodeGenerator->Writer()->DefineLabel();

EmitBooleanExpression(expr->sxBin.pnode1, leftTrue, falseLabel, byteCodeGenerator, funcInfo);

funcInfo->ReleaseLoc(expr->sxBin.pnode1);

byteCodeGenerator->Writer()->MarkLabel(leftTrue);

EmitBooleanExpression(expr->sxBin.pnode2, trueLabel, falseLabel, byteCodeGenerator, funcInfo);

funcInfo->ReleaseLoc(expr->sxBin.pnode2);

byteCodeGenerator->EndStatement(expr);

break;

}

case knopLogNot:

byteCodeGenerator->StartStatement(expr);

EmitBooleanExpression(expr->sxUni.pnode1, falseLabel, trueLabel, byteCodeGenerator, funcInfo);

funcInfo->ReleaseLoc(expr->sxUni.pnode1);

byteCodeGenerator->EndStatement(expr);

break;

case knopEq:

case knopEqv:

case knopNEqv:

case knopNe:

case knopLt:

case knopLe:

case knopGe:

case knopGt:

byteCodeGenerator->StartStatement(expr);

EmitBinaryOpnds(expr->sxBin.pnode1, expr->sxBin.pnode2, byteCodeGenerator, funcInfo);

funcInfo->ReleaseLoc(expr->sxBin.pnode2);

funcInfo->ReleaseLoc(expr->sxBin.pnode1);

byteCodeGenerator->Writer()->BrReg2(nopToOp[expr->nop], trueLabel, expr->sxBin.pnode1->location,

expr->sxBin.pnode2->location);

byteCodeGenerator->Writer()->Br(falseLabel);

byteCodeGenerator->EndStatement(expr);

break;

case knopTrue:

byteCodeGenerator->StartStatement(expr);

byteCodeGenerator->Writer()->Br(trueLabel);

byteCodeGenerator->EndStatement(expr);

break;

case knopFalse:

byteCodeGenerator->StartStatement(expr);

byteCodeGenerator->Writer()->Br(falseLabel);

byteCodeGenerator->EndStatement(expr);

break;

default:

// Note: we usually release the temp assigned to a node after we Emit it.

// But in this case, EmitBooleanExpression is just a wrapper around a normal Emit call,

// and the caller of EmitBooleanExpression expects to be able to release this register.

// For diagnostics purposes, register the name and dot to the statement list.

if (expr->nop == knopName || expr->nop == knopDot)

{

byteCodeGenerator->StartStatement(expr);

Emit(expr, byteCodeGenerator, funcInfo, false);

byteCodeGenerator->Writer()->BrReg1(Js::OpCode::BrTrue\_A, trueLabel, expr->location);

byteCodeGenerator->Writer()->Br(falseLabel);

byteCodeGenerator->EndStatement(expr);

}

else

{

Emit(expr, byteCodeGenerator, funcInfo, false);

byteCodeGenerator->Writer()->BrReg1(Js::OpCode::BrTrue\_A, trueLabel, expr->location);

byteCodeGenerator->Writer()->Br(falseLabel);

}

break;

}

}

// used by while and for loops

void EmitLoop(

ParseNode \*loopNode,

ParseNode \*cond,

ParseNode \*body,

ParseNode \*incr,

ByteCodeGenerator \*byteCodeGenerator,

FuncInfo \*funcInfo,

BOOL fReturnValue,

BOOL doWhile = FALSE,

ParseNode \*forLoopBlock = nullptr)

{

// Need to increment loop count whether we are going to profile or not for HasLoop()

Js::ByteCodeLabel loopEntrance = byteCodeGenerator->Writer()->DefineLabel();

Js::ByteCodeLabel continuePastLoop = byteCodeGenerator->Writer()->DefineLabel();

uint loopId = byteCodeGenerator->Writer()->EnterLoop(loopEntrance);

loopNode->sxLoop.loopId = loopId;

if (doWhile)

{

Emit(body, byteCodeGenerator, funcInfo, fReturnValue);

funcInfo->ReleaseLoc(body);

if (loopNode->emitLabels)

{

byteCodeGenerator->Writer()->MarkLabel(loopNode->sxStmt.continueLabel);

}

if (!ByteCodeGenerator::IsFalse(cond) ||

byteCodeGenerator->IsInDebugMode())

{

EmitBooleanExpression(cond, loopEntrance, continuePastLoop, byteCodeGenerator, funcInfo);

}

funcInfo->ReleaseLoc(cond);

}

else

{

if (cond)

{

if (!(cond->nop == knopInt &&

cond->sxInt.lw != 0))

{

Js::ByteCodeLabel trueLabel = byteCodeGenerator->Writer()->DefineLabel();

EmitBooleanExpression(cond, trueLabel, continuePastLoop, byteCodeGenerator, funcInfo);

byteCodeGenerator->Writer()->MarkLabel(trueLabel);

}

funcInfo->ReleaseLoc(cond);

}

Emit(body, byteCodeGenerator, funcInfo, fReturnValue);

funcInfo->ReleaseLoc(body);

if (byteCodeGenerator->IsES6ForLoopSemanticsEnabled() &&

forLoopBlock != nullptr)

{

CloneEmitBlock(forLoopBlock, byteCodeGenerator, funcInfo);

}

if (loopNode->emitLabels)

{

byteCodeGenerator->Writer()->MarkLabel(loopNode->sxStmt.continueLabel);

}

if (incr != nullptr)

{

Emit(incr, byteCodeGenerator, funcInfo, false);

funcInfo->ReleaseLoc(incr);

}

byteCodeGenerator->Writer()->Br(loopEntrance);

}

byteCodeGenerator->Writer()->MarkLabel(continuePastLoop);

if (loopNode->emitLabels)

{

byteCodeGenerator->Writer()->MarkLabel(loopNode->sxStmt.breakLabel);

}

byteCodeGenerator->Writer()->ExitLoop(loopId);

}

void ByteCodeGenerator::EmitInvertedLoop(ParseNode\* outerLoop, ParseNode\* invertedLoop, FuncInfo\* funcInfo)

{

Js::ByteCodeLabel invertedLoopLabel = this->m\_writer.DefineLabel();

Js::ByteCodeLabel afterInvertedLoop = this->m\_writer.DefineLabel();

// emit branch around original

Emit(outerLoop->sxFor.pnodeInit, this, funcInfo, false);

funcInfo->ReleaseLoc(outerLoop->sxFor.pnodeInit);

this->m\_writer.BrS(Js::OpCode::BrNotHasSideEffects, invertedLoopLabel, Js::SideEffects\_Any);

// emit original

EmitLoop(outerLoop, outerLoop->sxFor.pnodeCond, outerLoop->sxFor.pnodeBody,

outerLoop->sxFor.pnodeIncr, this, funcInfo, false);

// clear temporary registers since inverted loop may share nodes with

// emitted original loop

VisitClearTmpRegs(outerLoop, this, funcInfo);

// emit branch around inverted

this->m\_writer.Br(afterInvertedLoop);

this->m\_writer.MarkLabel(invertedLoopLabel);

// Emit a zero trip test for the original outer-loop

Js::ByteCodeLabel zeroTrip = this->m\_writer.DefineLabel();

ParseNode\* testNode = this->GetParser()->CopyPnode(outerLoop->sxFor.pnodeCond);

EmitBooleanExpression(testNode, zeroTrip, afterInvertedLoop, this, funcInfo);

this->m\_writer.MarkLabel(zeroTrip);

funcInfo->ReleaseLoc(testNode);

// emit inverted

Emit(invertedLoop->sxFor.pnodeInit, this, funcInfo, false);

funcInfo->ReleaseLoc(invertedLoop->sxFor.pnodeInit);

EmitLoop(invertedLoop, invertedLoop->sxFor.pnodeCond, invertedLoop->sxFor.pnodeBody,

invertedLoop->sxFor.pnodeIncr, this, funcInfo, false);

this->m\_writer.MarkLabel(afterInvertedLoop);

}

void EmitGetIterator(Js::RegSlot iteratorLocation, Js::RegSlot iterableLocation, ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* funcInfo)

{

// get iterator object from the iterable

EmitInvoke(iteratorLocation, iterableLocation, Js::PropertyIds::\_symbolIterator, byteCodeGenerator, funcInfo);

// throw TypeError if the result is not an object

Js::ByteCodeLabel skipThrow = byteCodeGenerator->Writer()->DefineLabel();

byteCodeGenerator->Writer()->BrReg1(Js::OpCode::BrOnObject\_A, skipThrow, iteratorLocation);

byteCodeGenerator->Writer()->W1(Js::OpCode::RuntimeTypeError, SCODE\_CODE(JSERR\_NeedObject));

byteCodeGenerator->Writer()->MarkLabel(skipThrow);

}

void EmitIteratorNext(Js::RegSlot itemLocation, Js::RegSlot iteratorLocation, Js::RegSlot nextInputLocation, ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* funcInfo)

{

// invoke next() on the iterator

if (nextInputLocation == Js::Constants::NoRegister)

{

EmitInvoke(itemLocation, iteratorLocation, Js::PropertyIds::next, byteCodeGenerator, funcInfo);

}

else

{

EmitInvoke(itemLocation, iteratorLocation, Js::PropertyIds::next, byteCodeGenerator, funcInfo, nextInputLocation);

}

// throw TypeError if the result is not an object

Js::ByteCodeLabel skipThrow = byteCodeGenerator->Writer()->DefineLabel();

byteCodeGenerator->Writer()->BrReg1(Js::OpCode::BrOnObject\_A, skipThrow, itemLocation);

byteCodeGenerator->Writer()->W1(Js::OpCode::RuntimeTypeError, SCODE\_CODE(JSERR\_NeedObject));

byteCodeGenerator->Writer()->MarkLabel(skipThrow);

}

void EmitIteratorComplete(Js::RegSlot doneLocation, Js::RegSlot iteratorResultLocation, ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* funcInfo)

{

// get the iterator result's "done" property

uint cacheId = funcInfo->FindOrAddInlineCacheId(iteratorResultLocation, Js::PropertyIds::done, false, false);

byteCodeGenerator->Writer()->PatchableProperty(Js::OpCode::LdFld, doneLocation, iteratorResultLocation, cacheId);

// Do not need to do ToBoolean explicitly with current uses of EmitIteratorComplete since BrTrue\_A does this.

// Add a ToBoolean controlled by template flag if needed for new uses later on.

}

void EmitIteratorValue(Js::RegSlot valueLocation, Js::RegSlot iteratorResultLocation, ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* funcInfo)

{

// get the iterator result's "value" property

uint cacheId = funcInfo->FindOrAddInlineCacheId(iteratorResultLocation, Js::PropertyIds::value, false, false);

byteCodeGenerator->Writer()->PatchableProperty(Js::OpCode::LdFld, valueLocation, iteratorResultLocation, cacheId);

}

void EmitForInOrForOf(ParseNode \*loopNode, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo, BOOL fReturnValue)

{

bool isForIn = (loopNode->nop == knopForIn);

Assert(isForIn || loopNode->nop == knopForOf);

BeginEmitBlock(loopNode->sxForInOrForOf.pnodeBlock, byteCodeGenerator, funcInfo);

byteCodeGenerator->StartStatement(loopNode);

funcInfo->AcquireLoc(loopNode);

// Record the branch bytecode offset.

// This is used for "ignore exception" and "set next stmt" scenarios. See ProbeContainer::GetNextUserStatementOffsetForAdvance:

// If there is a branch recorded between current offset and next stmt offset, we'll use offset of the branch recorded,

// otherwise use offset of next stmt.

// The idea here is that when we bail out after ignore exception, we need to bail out to the beginning of the ForIn,

// but currently ForIn stmt starts at the condition part, which is needed for correct handling of break point on ForIn

// (break every time on the loop back edge) and correct display of current statement under debugger.

// See WinBlue 231880 for details.

byteCodeGenerator->Writer()->RecordStatementAdjustment(Js::FunctionBody::SAT\_All);

if (byteCodeGenerator->IsES6ForLoopSemanticsEnabled() &&

loopNode->sxForInOrForOf.pnodeBlock->sxBlock.HasBlockScopedContent())

{

byteCodeGenerator->Writer()->RecordForInOrOfCollectionScope();

}

Js::ByteCodeLabel loopEntrance = byteCodeGenerator->Writer()->DefineLabel();

Js::ByteCodeLabel continuePastLoop = byteCodeGenerator->Writer()->DefineLabel();

Js::ByteCodeLabel skipPastLoop = byteCodeGenerator->Writer()->DefineLabel();

if (loopNode->sxForInOrForOf.pnodeLval->nop == knopVarDecl)

{

EmitReference(loopNode->sxForInOrForOf.pnodeLval, byteCodeGenerator, funcInfo);

}

Emit(loopNode->sxForInOrForOf.pnodeObj, byteCodeGenerator, funcInfo, false); // evaluate collection expression

funcInfo->ReleaseLoc(loopNode->sxForInOrForOf.pnodeObj);

if (byteCodeGenerator->IsES6ForLoopSemanticsEnabled())

{

EndEmitBlock(loopNode->sxForInOrForOf.pnodeBlock, byteCodeGenerator, funcInfo);

if (loopNode->sxForInOrForOf.pnodeBlock->sxBlock.scope != nullptr)

{

loopNode->sxForInOrForOf.pnodeBlock->sxBlock.scope->ForEachSymbol([](Symbol \*sym) {

sym->SetIsTrackedForDebugger(false);

});

}

}

// Grab registers for the enumerator and for the current enumerated item.

// The enumerator register will be released after this call returns.

loopNode->sxForInOrForOf.itemLocation = funcInfo->AcquireTmpRegister();

if (isForIn)

{

// get enumerator from the collection

byteCodeGenerator->Writer()->Reg2(Js::OpCode::GetForInEnumerator, loopNode->location, loopNode->sxForInOrForOf.pnodeObj->location);

}

else

{

// We want call profile information on the @@iterator call, so instead of adding a GetForOfIterator bytecode op

// to do all the following work in a helper do it explicitly in bytecode so that the @@iterator call is exposed

// to the profiler and JIT.

// If collection is null or undefined, don't enter the loop.

byteCodeGenerator->Writer()->BrReg2(Js::OpCode::BrSrEq\_A, skipPastLoop, loopNode->sxForInOrForOf.pnodeObj->location, funcInfo->nullConstantRegister);

byteCodeGenerator->Writer()->BrReg2(Js::OpCode::BrSrEq\_A, skipPastLoop, loopNode->sxForInOrForOf.pnodeObj->location, funcInfo->undefinedConstantRegister);

// do a ToObject on the collection

Js::RegSlot tmpObj = funcInfo->AcquireTmpRegister();

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Conv\_Obj, tmpObj, loopNode->sxForInOrForOf.pnodeObj->location);

EmitGetIterator(loopNode->location, tmpObj, byteCodeGenerator, funcInfo);

funcInfo->ReleaseTmpRegister(tmpObj);

}

byteCodeGenerator->EndStatement(loopNode);

// Need to increment loop count whether we are going into profile or not for HasLoop()

uint loopId = byteCodeGenerator->Writer()->EnterLoop(loopEntrance);

loopNode->sxForInOrForOf.loopId = loopId;

byteCodeGenerator->StartStatement(loopNode->sxForInOrForOf.pnodeLval);

if (isForIn)

{

// branch past loop when GetCurrentAndMoveNext returns nullptr

byteCodeGenerator->Writer()->BrReg2(Js::OpCode::BrOnEmpty, continuePastLoop, loopNode->sxForInOrForOf.itemLocation, loopNode->location);

}

else

{

EmitIteratorNext(loopNode->sxForInOrForOf.itemLocation, loopNode->location, Js::Constants::NoRegister, byteCodeGenerator, funcInfo);

Js::RegSlot doneLocation = funcInfo->AcquireTmpRegister();

EmitIteratorComplete(doneLocation, loopNode->sxForInOrForOf.itemLocation, byteCodeGenerator, funcInfo);

// branch past loop if the result's done property is truthy

byteCodeGenerator->Writer()->BrReg1(Js::OpCode::BrTrue\_A, continuePastLoop, doneLocation);

funcInfo->ReleaseTmpRegister(doneLocation);

// otherwise put result's value property in itemLocation

EmitIteratorValue(loopNode->sxForInOrForOf.itemLocation, loopNode->sxForInOrForOf.itemLocation, byteCodeGenerator, funcInfo);

}

if (loopNode->sxForInOrForOf.pnodeLval->nop != knopVarDecl &&

loopNode->sxForInOrForOf.pnodeLval->nop != knopLetDecl)

{

EmitReference(loopNode->sxForInOrForOf.pnodeLval, byteCodeGenerator, funcInfo);

}

else

{

Symbol \* sym = loopNode->sxForInOrForOf.pnodeLval->sxVar.sym;

sym->SetNeedDeclaration(false);

}

if (byteCodeGenerator->IsES6ForLoopSemanticsEnabled())

{

BeginEmitBlock(loopNode->sxForInOrForOf.pnodeBlock, byteCodeGenerator, funcInfo);

}

EmitAssignment(nullptr, loopNode->sxForInOrForOf.pnodeLval, loopNode->sxForInOrForOf.itemLocation, byteCodeGenerator, funcInfo);

byteCodeGenerator->EndStatement(loopNode->sxForInOrForOf.pnodeLval);

funcInfo->ReleaseReference(loopNode->sxForInOrForOf.pnodeLval);

Emit(loopNode->sxForInOrForOf.pnodeBody, byteCodeGenerator, funcInfo, fReturnValue);

funcInfo->ReleaseLoc(loopNode->sxForInOrForOf.pnodeBody);

if (byteCodeGenerator->IsES6ForLoopSemanticsEnabled())

{

EndEmitBlock(loopNode->sxForInOrForOf.pnodeBlock, byteCodeGenerator, funcInfo);

}

funcInfo->ReleaseTmpRegister(loopNode->sxForInOrForOf.itemLocation);

if (loopNode->emitLabels)

{

byteCodeGenerator->Writer()->MarkLabel(loopNode->sxForInOrForOf.continueLabel);

}

byteCodeGenerator->Writer()->Br(loopEntrance);

byteCodeGenerator->Writer()->MarkLabel(continuePastLoop);

if (loopNode->emitLabels)

{

byteCodeGenerator->Writer()->MarkLabel(loopNode->sxForInOrForOf.breakLabel);

}

byteCodeGenerator->Writer()->ExitLoop(loopId);

if (isForIn)

{

byteCodeGenerator->Writer()->Reg1(Js::OpCode::ReleaseForInEnumerator, loopNode->location);

}

else

{

byteCodeGenerator->Writer()->MarkLabel(skipPastLoop);

}

if (!byteCodeGenerator->IsES6ForLoopSemanticsEnabled())

{

EndEmitBlock(loopNode->sxForInOrForOf.pnodeBlock, byteCodeGenerator, funcInfo);

}

}

void EmitArrayLiteral(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo)

{

funcInfo->AcquireLoc(pnode);

ParseNode \*args = pnode->sxUni.pnode1;

if (args == nullptr)

{

byteCodeGenerator->Writer()->Reg1Unsigned1(

pnode->sxArrLit.hasMissingValues ? Js::OpCode::NewScArrayWithMissingValues : Js::OpCode::NewScArray,

pnode->location,

ByteCodeGenerator::DefaultArraySize);

}

else

{

SetNewArrayElements(pnode, pnode->location, byteCodeGenerator, funcInfo);

}

}

void EmitJumpCleanup(ParseNode \*pnode, ParseNode \*pnodeTarget, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \* funcInfo)

{

for (; pnode != pnodeTarget; pnode = pnode->sxStmt.pnodeOuter)

{

switch (pnode->nop)

{

case knopTry:

case knopCatch:

case knopFinally:

// We insert OpCode::Leave when there is a 'return' inside try/catch/finally.

// This is for flow control and does not participate in identifying boundaries of try/catch blocks,

// thus we shouldn't call RecordCrossFrameEntryExitRecord() here.

byteCodeGenerator->Writer()->Empty(Js::OpCode::Leave);

break;

#if ENABLE\_PROFILE\_INFO

case knopWhile:

case knopDoWhile:

case knopFor:

case knopForIn:

case knopForOf:

if (Js::DynamicProfileInfo::EnableImplicitCallFlags(funcInfo->GetParsedFunctionBody()))

{

byteCodeGenerator->Writer()->Unsigned1(Js::OpCode::ProfiledLoopEnd, pnode->sxLoop.loopId);

}

#endif

}

}

}

void EmitBinaryOpnds(ParseNode \*pnode1, ParseNode \*pnode2, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo)

{

// If opnd2 can overwrite opnd1, make sure the value of opnd1 is stashed away.

if (MayHaveSideEffectOnNode(pnode1, pnode2))

{

SaveOpndValue(pnode1, funcInfo);

}

Emit(pnode1, byteCodeGenerator, funcInfo, false);

if (pnode1->nop == knopComputedName && pnode2->nop == knopClassDecl &&

(pnode2->sxClass.pnodeConstructor == nullptr || pnode2->sxClass.pnodeConstructor->nop != knopVarDecl))

{

Emit(pnode2, byteCodeGenerator, funcInfo, false, false, pnode1);

}

else

{

Emit(pnode2, byteCodeGenerator, funcInfo, false);

}

}

void EmitBinaryReference(ParseNode \*pnode1, ParseNode \*pnode2, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo, BOOL fLoadLhs)

{

// Make sure that the RHS of an assignment doesn't kill the opnd's of the expression on the LHS.

switch (pnode1->nop)

{

case knopName:

if (fLoadLhs && MayHaveSideEffectOnNode(pnode1, pnode2))

{

// Given x op y, y may kill x, so stash x.

// Note that this only matters if we're loading x prior to the op.

SaveOpndValue(pnode1, funcInfo);

}

break;

case knopDot:

if (fLoadLhs)

{

// We're loading the value of the LHS before the RHS, so make sure the LHS gets a register first.

funcInfo->AcquireLoc(pnode1);

}

if (MayHaveSideEffectOnNode(pnode1->sxBin.pnode1, pnode2))

{

// Given x.y op z, z may kill x, so stash x away.

SaveOpndValue(pnode1->sxBin.pnode1, funcInfo);

}

break;

case knopIndex:

if (fLoadLhs)

{

// We're loading the value of the LHS before the RHS, so make sure the LHS gets a register first.

funcInfo->AcquireLoc(pnode1);

}

if (MayHaveSideEffectOnNode(pnode1->sxBin.pnode1, pnode2) ||

MayHaveSideEffectOnNode(pnode1->sxBin.pnode1, pnode1->sxBin.pnode2))

{

// Given x[y] op z, y or z may kill x, so stash x away.

SaveOpndValue(pnode1->sxBin.pnode1, funcInfo);

}

if (MayHaveSideEffectOnNode(pnode1->sxBin.pnode2, pnode2))

{

// Given x[y] op z, z may kill y, so stash y away.

// But make sure that x gets a register before y.

funcInfo->AcquireLoc(pnode1->sxBin.pnode1);

SaveOpndValue(pnode1->sxBin.pnode2, funcInfo);

}

break;

}

if (fLoadLhs)

{

// Emit code to load the value of the LHS.

EmitLoad(pnode1, byteCodeGenerator, funcInfo);

}

else

{

// Emit code to evaluate the LHS opnds, but don't load the LHS's value.

EmitReference(pnode1, byteCodeGenerator, funcInfo);

}

// Evaluate the RHS.

Emit(pnode2, byteCodeGenerator, funcInfo, false);

}

void EmitUseBeforeDeclarationRuntimeError(ByteCodeGenerator \* byteCodeGenerator, Js::RegSlot location, bool fLoadUndef)

{

byteCodeGenerator->Writer()->W1(Js::OpCode::RuntimeReferenceError, SCODE\_CODE(JSERR\_UseBeforeDeclaration));

// Load something into register in order to do not confuse IRBuilder. This value will never be used.

if (fLoadUndef)

{

byteCodeGenerator->Writer()->Reg1(Js::OpCode::LdUndef, location);

}

}

bool EmitUseBeforeDeclaration(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo)

{

Symbol \*sym = nullptr;

bool fAcquireLoc = true;

if (pnode->nop == knopName)

{

sym = pnode->sxPid.sym;

}

else if (pnode->nop == knopVarDecl)

{

fAcquireLoc = false;

sym = pnode->sxVar.sym;

}

else if ((ParseNode::Grfnop(pnode->nop) & fnopAsg) != 0)

{

if ((ParseNode::Grfnop(pnode->nop) & fnopBin) != 0 && pnode->sxBin.pnode1->nop == knopName)

{

sym = pnode->sxBin.pnode1->sxPid.sym;

}

if ((ParseNode::Grfnop(pnode->nop) & fnopUni) != 0 && pnode->sxUni.pnode1->nop == knopName)

{

sym = pnode->sxUni.pnode1->sxPid.sym;

}

}

// Don't emit static use-before-declaration error in a closure or dynamic scope case. We detect such cases with dynamic checks,

// if necessary.

if (sym != nullptr && sym->GetNeedDeclaration() && byteCodeGenerator->GetCurrentScope()->HasStaticPathToAncestor(sym->GetScope()) && sym->GetScope()->GetFunc() == funcInfo)

{

if (fAcquireLoc)

{

funcInfo->AcquireLoc(pnode);

}

EmitUseBeforeDeclarationRuntimeError(byteCodeGenerator, pnode->location, fAcquireLoc);

return true;

}

return false;

}

void EmitBinary(Js::OpCode opcode, ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo)

{

byteCodeGenerator->StartStatement(pnode);

EmitBinaryOpnds(pnode->sxBin.pnode1, pnode->sxBin.pnode2, byteCodeGenerator, funcInfo);

funcInfo->ReleaseLoc(pnode->sxBin.pnode2);

funcInfo->ReleaseLoc(pnode->sxBin.pnode1);

funcInfo->AcquireLoc(pnode);

byteCodeGenerator->Writer()->Reg3(opcode,

pnode->location,

pnode->sxBin.pnode1->location,

pnode->sxBin.pnode2->location);

byteCodeGenerator->EndStatement(pnode);

}

bool CollectConcat(ParseNode \*pnodeAdd, DListCounted<ParseNode \*, ArenaAllocator>& concatOpnds, ArenaAllocator \*arenaAllocator)

{

Assert(pnodeAdd->nop == knopAdd);

Assert(pnodeAdd->CanFlattenConcatExpr());

bool doConcatString = false;

DList<ParseNode\*, ArenaAllocator> pnodeStack(arenaAllocator);

pnodeStack.Prepend(pnodeAdd->sxBin.pnode2);

ParseNode \* pnode = pnodeAdd->sxBin.pnode1;

while (true)

{

if (!pnode->CanFlattenConcatExpr())

{

concatOpnds.Append(pnode);

}

else if (pnode->nop == knopStr)

{

concatOpnds.Append(pnode);

// Detect if there are any string larger then the append size limit.

// If there are, we can do concat; otherwise, still use add so we will not lose the AddLeftDead opportunities.

doConcatString = doConcatString || !Js::CompoundString::ShouldAppendChars(pnode->sxPid.pid->Cch());

}

else

{

Assert(pnode->nop == knopAdd);

pnodeStack.Prepend(pnode->sxBin.pnode2);

pnode = pnode->sxBin.pnode1;

continue;

}

if (pnodeStack.Empty())

{

break;

}

pnode = pnodeStack.Head();

pnodeStack.RemoveHead();

}

return doConcatString;

}

void EmitConcat3(ParseNode \*pnode, ParseNode \*pnode1, ParseNode \*pnode2, ParseNode \*pnode3, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo)

{

byteCodeGenerator->StartStatement(pnode);

if (MayHaveSideEffectOnNode(pnode1, pnode2) || MayHaveSideEffectOnNode(pnode1, pnode3))

{

SaveOpndValue(pnode1, funcInfo);

}

if (MayHaveSideEffectOnNode(pnode2, pnode3))

{

SaveOpndValue(pnode2, funcInfo);

}

Emit(pnode1, byteCodeGenerator, funcInfo, false);

Emit(pnode2, byteCodeGenerator, funcInfo, false);

Emit(pnode3, byteCodeGenerator, funcInfo, false);

funcInfo->ReleaseLoc(pnode3);

funcInfo->ReleaseLoc(pnode2);

funcInfo->ReleaseLoc(pnode1);

funcInfo->AcquireLoc(pnode);

byteCodeGenerator->Writer()->Reg4(Js::OpCode::Concat3,

pnode->location,

pnode1->location,

pnode2->location,

pnode3->location);

byteCodeGenerator->EndStatement(pnode);

}

void EmitNewConcatStrMulti(ParseNode \*pnode, uint8 count, ParseNode \*pnode1, ParseNode \*pnode2, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo)

{

EmitBinaryOpnds(pnode1, pnode2, byteCodeGenerator, funcInfo);

funcInfo->ReleaseLoc(pnode2);

funcInfo->ReleaseLoc(pnode1);

funcInfo->AcquireLoc(pnode);

byteCodeGenerator->Writer()->Reg3B1(Js::OpCode::NewConcatStrMulti,

pnode->location,

pnode1->location,

pnode2->location,

count);

}

void EmitAdd(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo)

{

Assert(pnode->nop == knopAdd);

if (pnode->CanFlattenConcatExpr())

{

// We should only have a string concat if the feature is on.

Assert(!PHASE\_OFF1(Js::ByteCodeConcatExprOptPhase));

DListCounted<ParseNode\*, ArenaAllocator> concatOpnds(byteCodeGenerator->GetAllocator());

bool doConcatString = CollectConcat(pnode, concatOpnds, byteCodeGenerator->GetAllocator());

if (doConcatString)

{

uint concatCount = concatOpnds.Count();

Assert(concatCount >= 2);

// Don't do concatN if the number is too high

// CONSIDER: although we could have done multiple ConcatNs

if (concatCount > 2 && concatCount <= UINT8\_MAX)

{

#if DBG

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

#endif

ParseNode \* pnode1 = concatOpnds.Head();

concatOpnds.RemoveHead();

ParseNode \* pnode2 = concatOpnds.Head();

concatOpnds.RemoveHead();

if (concatCount == 3)

{

OUTPUT\_TRACE\_DEBUGONLY(Js::ByteCodeConcatExprOptPhase, L"%s(%s) offset:#%d : Concat3\n",

funcInfo->GetParsedFunctionBody()->GetDisplayName(),

funcInfo->GetParsedFunctionBody()->GetDebugNumberSet(debugStringBuffer),

byteCodeGenerator->Writer()->ByteCodeDataSize());

EmitConcat3(pnode, pnode1, pnode2, concatOpnds.Head(), byteCodeGenerator, funcInfo);

return;

}

OUTPUT\_TRACE\_DEBUGONLY(Js::ByteCodeConcatExprOptPhase, L"%s(%s) offset:#%d: ConcatMulti %d\n",

funcInfo->GetParsedFunctionBody()->GetDisplayName(),

funcInfo->GetParsedFunctionBody()->GetDebugNumberSet(debugStringBuffer),

byteCodeGenerator->Writer()->ByteCodeDataSize(), concatCount);

byteCodeGenerator->StartStatement(pnode);

funcInfo->AcquireLoc(pnode);

// CONSIDER: this may cause the backend not able CSE repeating pattern within the concat.

EmitNewConcatStrMulti(pnode, (uint8)concatCount, pnode1, pnode2, byteCodeGenerator, funcInfo);

uint i = 2;

do

{

ParseNode \* currNode = concatOpnds.Head();

concatOpnds.RemoveHead();

ParseNode \* currNode2 = concatOpnds.Head();

concatOpnds.RemoveHead();

EmitBinaryOpnds(currNode, currNode2, byteCodeGenerator, funcInfo);

funcInfo->ReleaseLoc(currNode2);

funcInfo->ReleaseLoc(currNode);

byteCodeGenerator->Writer()->Reg3B1(

Js::OpCode::SetConcatStrMultiItem2, pnode->location, currNode->location, currNode2->location, (uint8)i);

i += 2;

}

while (concatOpnds.Count() > 1);

if (!concatOpnds.Empty())

{

ParseNode \* currNode = concatOpnds.Head();

Emit(currNode, byteCodeGenerator, funcInfo, false);

funcInfo->ReleaseLoc(currNode);

byteCodeGenerator->Writer()->Reg2B1(

Js::OpCode::SetConcatStrMultiItem, pnode->location, currNode->location, (uint8)i);

i++;

}

Assert(concatCount == i);

byteCodeGenerator->EndStatement(pnode);

return;

}

}

// Since we collected all the node already, let's just emit them instead of doing it recursively.

byteCodeGenerator->StartStatement(pnode);

ParseNode \* currNode = concatOpnds.Head();

concatOpnds.RemoveHead();

ParseNode \* currNode2 = concatOpnds.Head();

concatOpnds.RemoveHead();

EmitBinaryOpnds(currNode, currNode2, byteCodeGenerator, funcInfo);

funcInfo->ReleaseLoc(currNode2);

funcInfo->ReleaseLoc(currNode);

Js::RegSlot dstReg = funcInfo->AcquireLoc(pnode);

byteCodeGenerator->Writer()->Reg3(

Js::OpCode::Add\_A, dstReg, currNode->location, currNode2->location);

while (!concatOpnds.Empty())

{

currNode = concatOpnds.Head();

concatOpnds.RemoveHead();

Emit(currNode, byteCodeGenerator, funcInfo, false);

funcInfo->ReleaseLoc(currNode);

byteCodeGenerator->Writer()->Reg3(

Js::OpCode::Add\_A, dstReg, dstReg, currNode->location);

}

byteCodeGenerator->EndStatement(pnode);

}

else

{

EmitBinary(Js::OpCode::Add\_A, pnode, byteCodeGenerator, funcInfo);

}

}

void EmitSuperFieldPatch(FuncInfo\* funcInfo, ParseNode\* pnode, ByteCodeGenerator\* byteCodeGenerator)

{

ParseNodePtr propFuncNode = funcInfo->root;

if (byteCodeGenerator->GetFlags() & fscrEval)

{

// If we are inside an eval, ScopedLdSuper will take care of the patch.

return;

}

if (funcInfo->IsLambda())

{

FuncInfo \*parent = byteCodeGenerator->FindEnclosingNonLambda();

propFuncNode = parent->root;

}

// No need to emit a LdFld for the constructor.

if (propFuncNode->sxFnc.IsClassConstructor())

{

return;

}

if (!propFuncNode->sxFnc.IsClassMember() || propFuncNode->sxFnc.pid == nullptr)

{

// Non-methods will fail lookup.

return;

}

if (propFuncNode->sxFnc.pid->GetPropertyId() == Js::Constants::NoProperty)

{

byteCodeGenerator->AssignPropertyId(propFuncNode->sxFnc.pid);

}

// Load the current method's property ID from super instead of using super directly.

Js::RegSlot superLoc = funcInfo->superRegister;

pnode->sxCall.pnodeTarget->location = Js::Constants::NoRegister;

Js::RegSlot superPropLoc = funcInfo->AcquireLoc(pnode->sxCall.pnodeTarget);

Js::PropertyId propertyId = propFuncNode->sxFnc.pid->GetPropertyId();

uint cacheId = funcInfo->FindOrAddInlineCacheId(superLoc, propertyId, true, false);

byteCodeGenerator->Writer()->PatchableProperty(Js::OpCode::LdMethodFld, superPropLoc, superLoc, cacheId);

propFuncNode->sxFnc.pnodeName = nullptr;

}

struct ByteCodeGenerator::TryScopeRecord : public JsUtil::DoublyLinkedListElement<TryScopeRecord>

{

Js::OpCode op;

Js::ByteCodeLabel label;

Js::RegSlot reg1;

Js::RegSlot reg2;

TryScopeRecord(Js::OpCode op, Js::ByteCodeLabel label) : op(op), label(label), reg1(Js::Constants::NoRegister), reg2(Js::Constants::NoRegister) { }

TryScopeRecord(Js::OpCode op, Js::ByteCodeLabel label, Js::RegSlot r1, Js::RegSlot r2) : op(op), label(label), reg1(r1), reg2(r2) { }

};

void ByteCodeGenerator::EmitLeaveOpCodesBeforeYield()

{

for (TryScopeRecord\* node = this->tryScopeRecordsList.Tail(); node != nullptr; node = node->Previous())

{

switch (node->op)

{

case Js::OpCode::TryFinallyWithYield:

this->Writer()->Empty(Js::OpCode::LeaveNull);

break;

case Js::OpCode::TryCatch:

case Js::OpCode::ResumeFinally:

case Js::OpCode::ResumeCatch:

this->Writer()->Empty(Js::OpCode::Leave);

break;

default:

AssertMsg(false, "Unexpected OpCode before Yield in the Try-Catch-Finally cache for generator!");

break;

}

}

}

void ByteCodeGenerator::EmitTryBlockHeadersAfterYield()

{

for (TryScopeRecord\* node = this->tryScopeRecordsList.Head(); node != nullptr; node = node->Next())

{

switch (node->op)

{

case Js::OpCode::TryCatch:

this->Writer()->Br(node->op, node->label);

break;

case Js::OpCode::TryFinallyWithYield:

case Js::OpCode::ResumeFinally:

this->Writer()->BrReg2(node->op, node->label, node->reg1, node->reg2);

break;

case Js::OpCode::ResumeCatch:

this->Writer()->Empty(node->op);

break;

default:

AssertMsg(false, "Unexpected OpCode after yield in the Try-Catch-Finally cache for generator!");

break;

}

}

}

void EmitYield(Js::RegSlot inputLocation, Js::RegSlot resultLocation, ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* funcInfo,

Js::RegSlot yieldStarIterator = Js::Constants::NoRegister)

{

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, funcInfo->yieldRegister, inputLocation);

byteCodeGenerator->EmitLeaveOpCodesBeforeYield();

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Yield, funcInfo->yieldRegister, funcInfo->yieldRegister);

byteCodeGenerator->EmitTryBlockHeadersAfterYield();

if (yieldStarIterator == Js::Constants::NoRegister)

{

byteCodeGenerator->Writer()->Reg2(Js::OpCode::ResumeYield, resultLocation, funcInfo->yieldRegister);

}

else

{

byteCodeGenerator->Writer()->Reg3(Js::OpCode::ResumeYieldStar, resultLocation, funcInfo->yieldRegister, yieldStarIterator);

}

}

void EmitYieldStar(ParseNode\* yieldStarNode, ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* funcInfo)

{

funcInfo->AcquireLoc(yieldStarNode);

Js::ByteCodeLabel loopEntrance = byteCodeGenerator->Writer()->DefineLabel();

Js::ByteCodeLabel continuePastLoop = byteCodeGenerator->Writer()->DefineLabel();

Js::RegSlot iteratorLocation = funcInfo->AcquireTmpRegister();

// Evaluate operand

Emit(yieldStarNode->sxUni.pnode1, byteCodeGenerator, funcInfo, false);

funcInfo->ReleaseLoc(yieldStarNode->sxUni.pnode1);

EmitGetIterator(iteratorLocation, yieldStarNode->sxUni.pnode1->location, byteCodeGenerator, funcInfo);

uint loopId = byteCodeGenerator->Writer()->EnterLoop(loopEntrance);

// since a yield\* doesn't have a user defined body, we cannot return from this loop

// which means we don't need to support EmitJumpCleanup() and there do not need to

// remember the loopId like the loop statements do.

// Get a temporary to hold the input for the next() calls. Initialize it to undefined for the first call.

Js::RegSlot nextInputLocation = funcInfo->AcquireTmpRegister();

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, nextInputLocation, funcInfo->undefinedConstantRegister);

// Call the iterator's next()

EmitIteratorNext(yieldStarNode->location, iteratorLocation, nextInputLocation, byteCodeGenerator, funcInfo);

Js::RegSlot doneLocation = funcInfo->AcquireTmpRegister();

EmitIteratorComplete(doneLocation, yieldStarNode->location, byteCodeGenerator, funcInfo);

// Put the iterator result's value in yieldStarNode->location regardless of the done property value. The value will

// be used as the result value of the yield\* operator expression in the case when done is true.

EmitIteratorValue(yieldStarNode->location, yieldStarNode->location, byteCodeGenerator, funcInfo);

// branch past the loop if the done property is truthy

byteCodeGenerator->Writer()->BrReg1(Js::OpCode::BrTrue\_A, continuePastLoop, doneLocation);

funcInfo->ReleaseTmpRegister(doneLocation);

EmitYield(yieldStarNode->location, nextInputLocation, byteCodeGenerator, funcInfo, iteratorLocation);

funcInfo->ReleaseTmpRegister(nextInputLocation);

funcInfo->ReleaseTmpRegister(iteratorLocation);

byteCodeGenerator->Writer()->Br(loopEntrance);

byteCodeGenerator->Writer()->MarkLabel(continuePastLoop);

byteCodeGenerator->Writer()->ExitLoop(loopId);

}

void TrackIntConstantsOnGlobalUserObject(ByteCodeGenerator \*byteCodeGenerator, bool isSymGlobalAndSingleAssignment, Js::PropertyId propertyId)

{

if (isSymGlobalAndSingleAssignment)

{

byteCodeGenerator->GetScriptContext()->TrackIntConstPropertyOnGlobalUserObject(propertyId);

}

}

void TrackIntConstantsOnGlobalObject(ByteCodeGenerator \*byteCodeGenerator, bool isSymGlobalAndSingleAssignment, Js::PropertyId propertyId)

{

if (isSymGlobalAndSingleAssignment)

{

byteCodeGenerator->GetScriptContext()->TrackIntConstPropertyOnGlobalObject(propertyId);

}

}

void TrackIntConstantsOnGlobalObject(ByteCodeGenerator \*byteCodeGenerator, Symbol \*sym)

{

if (sym && sym->GetIsGlobal() && sym->IsAssignedOnce())

{

Js::PropertyId propertyId = sym->EnsurePosition(byteCodeGenerator);

byteCodeGenerator->GetScriptContext()->TrackIntConstPropertyOnGlobalObject(propertyId);

}

}

void TrackMemberNodesInObjectForIntConstants(ByteCodeGenerator \*byteCodeGenerator, ParseNodePtr objNode)

{

Assert(objNode->nop == knopObject);

Assert(ParseNode::Grfnop(objNode->nop) & fnopUni);

ParseNodePtr memberList = objNode->sxUni.pnode1;

if (memberList != nullptr)

{

// Iterate through all the member nodes

while (memberList->nop == knopList)

{

ParseNodePtr memberNode = memberList->sxBin.pnode1;

ParseNodePtr memberNameNode = memberNode->sxBin.pnode1;

ParseNodePtr memberValNode = memberNode->sxBin.pnode2;

if (memberNameNode->nop != knopComputedName && memberValNode->nop == knopInt)

{

Js::PropertyId propertyId = memberNameNode->sxPid.PropertyIdFromNameNode();

TrackIntConstantsOnGlobalUserObject(byteCodeGenerator, true, propertyId);

}

memberList = memberList->sxBin.pnode2;

}

ParseNode \*memberNameNode = memberList->sxBin.pnode1;

ParseNode \*memberValNode = memberList->sxBin.pnode2;

if (memberNameNode->nop != knopComputedName && memberValNode->nop == knopInt)

{

Js::PropertyId propertyId = memberNameNode->sxPid.PropertyIdFromNameNode();

TrackIntConstantsOnGlobalUserObject(byteCodeGenerator, true, propertyId);

}

}

}

void TrackGlobalIntAssignmentsForknopDotProps(ParseNodePtr knopDotNode, ByteCodeGenerator \* byteCodeGenerator)

{

Assert(knopDotNode->nop == knopDot);

ParseNodePtr objectNode = knopDotNode->sxBin.pnode1;

ParseNodePtr propertyNode = knopDotNode->sxBin.pnode2;

bool isSymGlobalAndSingleAssignment = false;

if (objectNode->nop == knopName)

{

Symbol \* sym = objectNode->sxVar.sym;

isSymGlobalAndSingleAssignment = sym && sym->GetIsGlobal() && sym->IsAssignedOnce() && propertyNode->sxPid.pid->IsSingleAssignment();

Js::PropertyId propertyId = propertyNode->sxPid.PropertyIdFromNameNode();

TrackIntConstantsOnGlobalUserObject(byteCodeGenerator, isSymGlobalAndSingleAssignment, propertyId);

}

else if (objectNode->nop == knopThis)

{

// Assume knopThis always refer to GlobalObject

// Cases like "this.a = "

isSymGlobalAndSingleAssignment = propertyNode->sxPid.pid->IsSingleAssignment();

Js::PropertyId propertyId = propertyNode->sxPid.PropertyIdFromNameNode();

TrackIntConstantsOnGlobalObject(byteCodeGenerator, isSymGlobalAndSingleAssignment, propertyId);

}

}

void TrackGlobalIntAssignments(ParseNodePtr pnode, ByteCodeGenerator \* byteCodeGenerator)

{

// Track the Global Int Constant properties' assignments here.

uint nodeType = ParseNode::Grfnop(pnode->nop);

if (nodeType & fnopAsg)

{

if (nodeType & fnopBin)

{

ParseNodePtr lhs = pnode->sxBin.pnode1;

ParseNodePtr rhs = pnode->sxBin.pnode2;

Assert(lhs && rhs);

// Don't track other than integers and objects with member nodes.

if (rhs->nop == knopObject && (ParseNode::Grfnop(rhs->nop) & fnopUni))

{

TrackMemberNodesInObjectForIntConstants(byteCodeGenerator, rhs);

}

else if (rhs->nop != knopInt &&

((rhs->nop != knopLsh && rhs->nop != knopRsh) || (rhs->sxBin.pnode1->nop != knopInt || rhs->sxBin.pnode2->nop != knopInt)))

{

return;

}

if (lhs->nop == knopName)

{

// Handle "a = <Integer>" cases here

Symbol \* sym = lhs->sxVar.sym;

TrackIntConstantsOnGlobalObject(byteCodeGenerator, sym);

}

else if (lhs->nop == knopDot && lhs->sxBin.pnode2->nop == knopName)

{

// Cases like "obj.a = <Integer>"

TrackGlobalIntAssignmentsForknopDotProps(lhs, byteCodeGenerator);

}

}

else if (nodeType & fnopUni)

{

ParseNodePtr lhs = pnode->sxUni.pnode1;

if (lhs->nop == knopName)

{

// Cases like "a++"

Symbol \* sym = lhs->sxVar.sym;

TrackIntConstantsOnGlobalObject(byteCodeGenerator, sym);

}

else if (lhs->nop == knopDot && lhs->sxBin.pnode2->nop == knopName)

{

// Cases like "obj.a++"

TrackGlobalIntAssignmentsForknopDotProps(lhs, byteCodeGenerator);

}

}

}

}

void Emit(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo, BOOL fReturnValue, bool isConstructorCall, ParseNode \* bindPnode)

{

if (pnode == nullptr)

{

return;

}

if (EmitUseBeforeDeclaration(pnode, byteCodeGenerator, funcInfo))

{

if (fReturnValue && IsExpressionStatement(pnode, byteCodeGenerator->GetScriptContext()))

{

// If this statement may produce the global function's return value, make sure we load something to the

// return register for the JIT.

// fReturnValue implies global function, which implies that "return" is a parse error.

Assert(funcInfo->IsGlobalFunction());

Assert(pnode->nop != knopReturn);

byteCodeGenerator->Writer()->Reg1(Js::OpCode::LdUndef, ByteCodeGenerator::ReturnRegister);

}

return;

}

ThreadContext::ProbeCurrentStackNoDispose(Js::Constants::MinStackByteCodeVisitor, byteCodeGenerator->GetScriptContext());

TrackGlobalIntAssignments(pnode, byteCodeGenerator);

// printNop(pnode->nop);

switch (pnode->nop)

{

case knopList:

EmitList(pnode, byteCodeGenerator, funcInfo);

break;

case knopInt:

// currently, these are loaded at the top

break;

// PTNODE(knopFlt , "flt const" ,None ,Flt ,fnopLeaf|fnopConst)

case knopFlt:

// currently, these are loaded at the top

break;

// PTNODE(knopStr , "str const" ,None ,Pid ,fnopLeaf|fnopConst)

case knopStr:

// TODO: protocol for combining string constants

break;

// PTNODE(knopRegExp , "reg expr" ,None ,Pid ,fnopLeaf|fnopConst)

case knopRegExp:

funcInfo->GetParsedFunctionBody()->SetLiteralRegex(pnode->sxPid.regexPatternIndex, pnode->sxPid.regexPattern);

byteCodeGenerator->Writer()->Reg1Unsigned1(Js::OpCode::NewRegEx, funcInfo->AcquireLoc(pnode), pnode->sxPid.regexPatternIndex);

break; // PTNODE(knopThis , "this" ,None ,None ,fnopLeaf)

case knopThis:

// enregistered

// Try to load 'this' from a scope slot if we are in a derived class constructor with scope slots. Otherwise, this is a nop.

byteCodeGenerator->EmitScopeSlotLoadThis(funcInfo, funcInfo->thisPointerRegister);

break;

// PTNODE(knopNewTarget , "new.target" ,None , None , fnopLeaf)

case knopNewTarget:

break;

// PTNODE(knopSuper , "super" ,None , None , fnopLeaf)

case knopSuper:

if (!funcInfo->IsClassMember())

{

FuncInfo\* nonLambdaFunc = funcInfo;

if (funcInfo->IsLambda())

{

nonLambdaFunc = byteCodeGenerator->FindEnclosingNonLambda();

}

if (nonLambdaFunc->IsGlobalFunction())

{

if ((byteCodeGenerator->GetFlags() & fscrEval))

{

byteCodeGenerator->Writer()->Reg1(isConstructorCall ? Js::OpCode::ScopedLdSuperCtor : Js::OpCode::ScopedLdSuper, funcInfo->AcquireLoc(pnode));

}

else

{

byteCodeGenerator->Writer()->W1(Js::OpCode::RuntimeReferenceError, SCODE\_CODE(JSERR\_BadSuperReference));

}

}

else

{

Js::ByteCodeLabel errLabel = byteCodeGenerator->Writer()->DefineLabel();

Js::ByteCodeLabel skipLabel = byteCodeGenerator->Writer()->DefineLabel();

byteCodeGenerator->Writer()->BrReg2(Js::OpCode::BrEq\_A, errLabel, funcInfo->superRegister, funcInfo->undefinedConstantRegister);

byteCodeGenerator->Writer()->Br(Js::OpCode::Br, skipLabel);

byteCodeGenerator->Writer()->MarkLabel(errLabel);

byteCodeGenerator->Writer()->W1(Js::OpCode::RuntimeReferenceError, SCODE\_CODE(JSERR\_BadSuperReference));

byteCodeGenerator->Writer()->MarkLabel(skipLabel);

}

}

break;

// PTNODE(knopNull , "null" ,Null ,None ,fnopLeaf)

case knopNull:

// enregistered

break;

// PTNODE(knopFalse , "false" ,False ,None ,fnopLeaf)

case knopFalse:

// enregistered

break;

// PTNODE(knopTrue , "true" ,True ,None ,fnopLeaf)

case knopTrue:

// enregistered

break;

// PTNODE(knopEmpty , "empty" ,Empty ,None ,fnopLeaf)

case knopEmpty:

break;

// Unary operators.

// PTNODE(knopNot , "~" ,BitNot ,Uni ,fnopUni)

case knopNot:

Emit(pnode->sxUni.pnode1, byteCodeGenerator, funcInfo, false);

funcInfo->ReleaseLoc(pnode->sxUni.pnode1);

byteCodeGenerator->Writer()->Reg2(

Js::OpCode::Not\_A, funcInfo->AcquireLoc(pnode), pnode->sxUni.pnode1->location);

break;

// PTNODE(knopNeg , "unary -" ,Neg ,Uni ,fnopUni)

case knopNeg:

Emit(pnode->sxUni.pnode1, byteCodeGenerator, funcInfo, false);

funcInfo->ReleaseLoc(pnode->sxUni.pnode1);

funcInfo->AcquireLoc(pnode);

byteCodeGenerator->Writer()->Reg2(

Js::OpCode::Neg\_A, pnode->location, pnode->sxUni.pnode1->location);

break;

// PTNODE(knopPos , "unary +" ,Pos ,Uni ,fnopUni)

case knopPos:

Emit(pnode->sxUni.pnode1, byteCodeGenerator, funcInfo, false);

funcInfo->ReleaseLoc(pnode->sxUni.pnode1);

byteCodeGenerator->Writer()->Reg2(

Js::OpCode::Conv\_Num, funcInfo->AcquireLoc(pnode), pnode->sxUni.pnode1->location);

break;

// PTNODE(knopLogNot , "!" ,LogNot ,Uni ,fnopUni)

case knopLogNot:

{

Js::ByteCodeLabel doneLabel = byteCodeGenerator->Writer()->DefineLabel();

// For boolean expressions that compute a result, we have to burn a register for the result

// so that the back end can identify it cheaply as a single temp lifetime. Revisit this if we do

// full-on renaming in the back end.

funcInfo->AcquireLoc(pnode);

if (pnode->sxUni.pnode1->nop == knopInt)

{

long value = pnode->sxUni.pnode1->sxInt.lw;

Js::OpCode op = value ? Js::OpCode::LdFalse : Js::OpCode::LdTrue;

byteCodeGenerator->Writer()->Reg1(op, pnode->location);

}

else

{

Emit(pnode->sxUni.pnode1, byteCodeGenerator, funcInfo, false);

byteCodeGenerator->Writer()->Reg1(Js::OpCode::LdFalse, pnode->location);

byteCodeGenerator->Writer()->BrReg1(Js::OpCode::BrTrue\_A, doneLabel, pnode->sxUni.pnode1->location);

byteCodeGenerator->Writer()->Reg1(Js::OpCode::LdTrue, pnode->location);

byteCodeGenerator->Writer()->MarkLabel(doneLabel);

}

funcInfo->ReleaseLoc(pnode->sxUni.pnode1);

break;

}

// PTNODE(knopEllipsis , "..." ,Spread ,Uni , fnopUni)

case knopEllipsis:

{

Emit(pnode->sxUni.pnode1, byteCodeGenerator, funcInfo, false);

// Transparently pass the location of the array.

pnode->location = pnode->sxUni.pnode1->location;

break;

}

// PTNODE(knopIncPost , "post++" ,Inc ,Uni ,fnopUni|fnopAsg)

case knopIncPost:

case knopDecPost:

// FALL THROUGH to the faster pre-inc/dec case if the result of the expression is not needed.

if (pnode->isUsed || fReturnValue)

{

byteCodeGenerator->StartStatement(pnode);

Js::OpCode op = Js::OpCode::Add\_A;

if (pnode->nop == knopDecPost)

{

op = Js::OpCode::Sub\_A;

}

// Grab a register for the expression result.

funcInfo->AcquireLoc(pnode);

// Load the initial value, convert it (this is the expression result), and increment it.

EmitLoad(pnode->sxUni.pnode1, byteCodeGenerator, funcInfo);

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Conv\_Num, pnode->location, pnode->sxUni.pnode1->location);

Js::RegSlot incDecResult = pnode->sxUni.pnode1->location;

if (funcInfo->RegIsConst(incDecResult))

{

// Avoid letting the add/sub overwrite a constant reg, as this may actually change the

// contents of the constant table.

incDecResult = funcInfo->AcquireTmpRegister();

}

Js::RegSlot oneReg = funcInfo->constantToRegister.LookupWithKey(1, Js::Constants::NoRegister);

Assert(oneReg != Js::Constants::NoRegister);

byteCodeGenerator->Writer()->Reg3(op, incDecResult, pnode->location, oneReg);

// Store the incremented value.

EmitAssignment(nullptr, pnode->sxUni.pnode1, incDecResult, byteCodeGenerator, funcInfo);

// Release the incremented value and the l-value.

if (incDecResult != pnode->sxUni.pnode1->location)

{

funcInfo->ReleaseTmpRegister(incDecResult);

}

funcInfo->ReleaseLoad(pnode->sxUni.pnode1);

byteCodeGenerator->EndStatement(pnode);

break;

}

else

{

pnode->nop = (pnode->nop == knopIncPost) ? knopIncPre : knopDecPre;

}

// FALL THROUGH to the fast pre-inc/dec case if the result of the expression is not needed.

// PTNODE(knopIncPre , "++ pre" ,Inc ,Uni ,fnopUni|fnopAsg)

case knopIncPre:

case knopDecPre:

{

byteCodeGenerator->StartStatement(pnode);

Js::OpCode op = Js::OpCode::Incr\_A;

if (pnode->nop == knopDecPre)

{

op = Js::OpCode::Decr\_A;

}

// Assign a register for the result only if the result is used or the operand can't be assigned to

// (i.e., is a constant).

if (pnode->isUsed || fReturnValue)

{

funcInfo->AcquireLoc(pnode);

// Load the initial value and increment it (this is the expression result).

EmitLoad(pnode->sxUni.pnode1, byteCodeGenerator, funcInfo);

byteCodeGenerator->Writer()->Reg2(op, pnode->location, pnode->sxUni.pnode1->location);

// Store the incremented value and release the l-value.

EmitAssignment(nullptr, pnode->sxUni.pnode1, pnode->location, byteCodeGenerator, funcInfo);

funcInfo->ReleaseLoad(pnode->sxUni.pnode1);

}

else

{

// Load the initial value and increment it (this is the expression result).

EmitLoad(pnode->sxUni.pnode1, byteCodeGenerator, funcInfo);

Js::RegSlot incDecResult = pnode->sxUni.pnode1->location;

if (funcInfo->RegIsConst(incDecResult))

{

// Avoid letting the add/sub overwrite a constant reg, as this may actually change the

// contents of the constant table.

incDecResult = funcInfo->AcquireTmpRegister();

}

byteCodeGenerator->Writer()->Reg2(op, incDecResult, pnode->sxUni.pnode1->location);

// Store the incremented value and release the l-value.

EmitAssignment(nullptr, pnode->sxUni.pnode1, incDecResult, byteCodeGenerator, funcInfo);

if (incDecResult != pnode->sxUni.pnode1->location)

{

funcInfo->ReleaseTmpRegister(incDecResult);

}

funcInfo->ReleaseLoad(pnode->sxUni.pnode1);

}

byteCodeGenerator->EndStatement(pnode);

break;

}

// PTNODE(knopTypeof , "typeof" ,None ,Uni ,fnopUni)

case knopTypeof:

{

ParseNode\* pnodeOpnd = pnode->sxUni.pnode1;

switch (pnodeOpnd->nop)

{

case knopDot:

{

Emit(pnodeOpnd->sxBin.pnode1, byteCodeGenerator, funcInfo, false);

Js::PropertyId propertyId = pnodeOpnd->sxBin.pnode2->sxPid.PropertyIdFromNameNode();

Assert(pnodeOpnd->sxBin.pnode2->nop == knopName);

funcInfo->ReleaseLoc(pnodeOpnd->sxBin.pnode1);

funcInfo->AcquireLoc(pnode);

byteCodeGenerator->EmitTypeOfFld(funcInfo, propertyId, pnode->location, pnodeOpnd->sxBin.pnode1->location, Js::OpCode::LdFldForTypeOf);

break;

}

case knopIndex:

{

EmitBinaryOpnds(pnodeOpnd->sxBin.pnode1, pnodeOpnd->sxBin.pnode2, byteCodeGenerator, funcInfo);

funcInfo->ReleaseLoc(pnodeOpnd->sxBin.pnode2);

funcInfo->ReleaseLoc(pnodeOpnd->sxBin.pnode1);

funcInfo->AcquireLoc(pnode);

byteCodeGenerator->Writer()->Element(Js::OpCode::TypeofElem, pnode->location, pnodeOpnd->sxBin.pnode1->location, pnodeOpnd->sxBin.pnode2->location);

break;

}

case knopName:

{

funcInfo->AcquireLoc(pnode);

byteCodeGenerator->EmitPropTypeof(pnode->location, pnodeOpnd->sxPid.sym, pnodeOpnd->sxPid.pid, funcInfo);

break;

}

default:

Emit(pnodeOpnd, byteCodeGenerator, funcInfo, false);

funcInfo->ReleaseLoc(pnodeOpnd);

byteCodeGenerator->Writer()->Reg2(

Js::OpCode::Typeof, funcInfo->AcquireLoc(pnode), pnodeOpnd->location);

break;

}

break;

}

// PTNODE(knopVoid , "void" ,Void ,Uni ,fnopUni)

case knopVoid:

Emit(pnode->sxUni.pnode1, byteCodeGenerator, funcInfo, false);

funcInfo->ReleaseLoc(pnode->sxUni.pnode1);

byteCodeGenerator->Writer()->Reg1(Js::OpCode::LdUndef, funcInfo->AcquireLoc(pnode));

break;

// PTNODE(knopArray , "arr cnst" ,None ,Uni ,fnopUni)

case knopArray:

EmitArrayLiteral(pnode, byteCodeGenerator, funcInfo);

break;

// PTNODE(knopObject , "obj cnst" ,None ,Uni ,fnopUni)

case knopObject:

funcInfo->AcquireLoc(pnode);

EmitObjectInitializers(pnode->sxUni.pnode1, pnode->location, byteCodeGenerator, funcInfo);

break;

// PTNODE(knopComputedName, "[name]" ,None ,Uni ,fnopUni)

case knopComputedName:

Emit(pnode->sxUni.pnode1, byteCodeGenerator, funcInfo, false);

if (pnode->location == Js::Constants::NoRegister)

{

// The name is some expression with no home location. We can just re-use the register.

pnode->location = pnode->sxUni.pnode1->location;

}

else if (pnode->location != pnode->sxUni.pnode1->location)

{

// The name had to be protected from side-effects of the RHS.

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, pnode->location, pnode->sxUni.pnode1->location);

}

break;

// Binary and Ternary Operators

case knopAdd:

EmitAdd(pnode, byteCodeGenerator, funcInfo);

break;

case knopSub:

case knopMul:

case knopExpo:

case knopDiv:

case knopMod:

case knopOr:

case knopXor:

case knopAnd:

case knopLsh:

case knopRsh:

case knopRs2:

case knopIn:

EmitBinary(nopToOp[pnode->nop], pnode, byteCodeGenerator, funcInfo);

break;

case knopInstOf:

{

byteCodeGenerator->StartStatement(pnode);

EmitBinaryOpnds(pnode->sxBin.pnode1, pnode->sxBin.pnode2, byteCodeGenerator, funcInfo);

funcInfo->ReleaseLoc(pnode->sxBin.pnode2);

funcInfo->ReleaseLoc(pnode->sxBin.pnode1);

funcInfo->AcquireLoc(pnode);

uint cacheId = funcInfo->NewIsInstInlineCache();

byteCodeGenerator->Writer()->Reg3C(nopToOp[pnode->nop], pnode->location, pnode->sxBin.pnode1->location,

pnode->sxBin.pnode2->location, cacheId);

byteCodeGenerator->EndStatement(pnode);

}

break;

case knopEq:

case knopEqv:

case knopNEqv:

case knopNe:

case knopLt:

case knopLe:

case knopGe:

case knopGt:

byteCodeGenerator->StartStatement(pnode);

EmitBinaryOpnds(pnode->sxBin.pnode1, pnode->sxBin.pnode2, byteCodeGenerator, funcInfo);

funcInfo->ReleaseLoc(pnode->sxBin.pnode2);

funcInfo->ReleaseLoc(pnode->sxBin.pnode1);

funcInfo->AcquireLoc(pnode);

byteCodeGenerator->Writer()->Reg3(nopToCMOp[pnode->nop], pnode->location, pnode->sxBin.pnode1->location,

pnode->sxBin.pnode2->location);

byteCodeGenerator->EndStatement(pnode);

break;

case knopNew:

{

Js::ArgSlot argCount = pnode->sxCall.argCount;

argCount++; // include "this"

BOOL fSideEffectArgs = FALSE;

unsigned int tmpCount = CountArguments(pnode->sxCall.pnodeArgs, &fSideEffectArgs);

Assert(argCount == tmpCount);

if (argCount != (Js::ArgSlot)argCount)

{

Js::Throw::OutOfMemory();

}

byteCodeGenerator->StartStatement(pnode);

// Start call, allocate out param space

funcInfo->StartRecordingOutArgs(argCount);

// Assign the call target operand(s), putting them into expression temps if necessary to protect

// them from side-effects.

if (fSideEffectArgs)

{

SaveOpndValue(pnode->sxCall.pnodeTarget, funcInfo);

}

if (pnode->sxCall.pnodeTarget->nop == knopSuper)

{

EmitSuperFieldPatch(funcInfo, pnode, byteCodeGenerator);

}

Emit(pnode->sxCall.pnodeTarget, byteCodeGenerator, funcInfo, false, true);

if (pnode->sxCall.pnodeArgs == nullptr)

{

funcInfo->ReleaseLoc(pnode->sxCall.pnodeTarget);

Js::OpCode op = (CreateNativeArrays(byteCodeGenerator, funcInfo)

&& CallTargetIsArray(pnode->sxCall.pnodeTarget))

? Js::OpCode::NewScObjArray : Js::OpCode::NewScObject;

Assert(argCount == 1);

Js::ProfileId callSiteId = byteCodeGenerator->GetNextCallSiteId(op);

byteCodeGenerator->Writer()->StartCall(Js::OpCode::StartCall, argCount);

byteCodeGenerator->Writer()->CallI(op, funcInfo->AcquireLoc(pnode),

pnode->sxCall.pnodeTarget->location, argCount, callSiteId);

}

else

{

byteCodeGenerator->Writer()->StartCall(Js::OpCode::StartCall, argCount);

uint32 actualArgCount = 0;

if (IsCallOfConstants(pnode))

{

funcInfo->ReleaseLoc(pnode->sxCall.pnodeTarget);

actualArgCount = EmitNewObjectOfConstants(pnode, byteCodeGenerator, funcInfo, argCount);

}

else

{

Js::OpCode op;

if ((CreateNativeArrays(byteCodeGenerator, funcInfo) && CallTargetIsArray(pnode->sxCall.pnodeTarget)))

{

op = pnode->sxCall.spreadArgCount > 0 ? Js::OpCode::NewScObjArraySpread : Js::OpCode::NewScObjArray;

}

else

{

op = pnode->sxCall.spreadArgCount > 0 ? Js::OpCode::NewScObjectSpread : Js::OpCode::NewScObject;

}

Js::ProfileId callSiteId = byteCodeGenerator->GetNextCallSiteId(op);

Js::AuxArray<uint32> \*spreadIndices = nullptr;

actualArgCount = EmitArgList(pnode->sxCall.pnodeArgs, Js::Constants::NoRegister, Js::Constants::NoRegister, Js::Constants::NoRegister,

false, true, byteCodeGenerator, funcInfo, callSiteId, pnode->sxCall.spreadArgCount, &spreadIndices);

funcInfo->ReleaseLoc(pnode->sxCall.pnodeTarget);

if (pnode->sxCall.spreadArgCount > 0)

{

Assert(spreadIndices != nullptr);

uint spreadExtraAlloc = spreadIndices->count \* sizeof(uint32);

uint spreadIndicesSize = sizeof(\*spreadIndices) + spreadExtraAlloc;

byteCodeGenerator->Writer()->CallIExtended(op, funcInfo->AcquireLoc(pnode), pnode->sxCall.pnodeTarget->location,

(uint16)actualArgCount, Js::CallIExtended\_SpreadArgs,

spreadIndices, spreadIndicesSize, callSiteId);

}

else

{

byteCodeGenerator->Writer()->CallI(op, funcInfo->AcquireLoc(pnode), pnode->sxCall.pnodeTarget->location,

(uint16)actualArgCount, callSiteId);

}

}

Assert(argCount == actualArgCount);

}

// End call, pop param space

funcInfo->EndRecordingOutArgs(argCount);

byteCodeGenerator->EndStatement(pnode);

break;

}

case knopDelete:

{

ParseNode \*pexpr = pnode->sxUni.pnode1;

byteCodeGenerator->StartStatement(pnode);

switch (pexpr->nop)

{

case knopName:

{

funcInfo->AcquireLoc(pnode);

byteCodeGenerator->EmitPropDelete(pnode->location, pexpr->sxPid.sym, pexpr->sxPid.pid, funcInfo);

break;

}

case knopDot:

{

Emit(pexpr->sxBin.pnode1, byteCodeGenerator, funcInfo, false);

Js::PropertyId propertyId = pexpr->sxBin.pnode2->sxPid.PropertyIdFromNameNode();

funcInfo->ReleaseLoc(pexpr->sxBin.pnode1);

funcInfo->AcquireLoc(pnode);

byteCodeGenerator->Writer()->Property(Js::OpCode::DeleteFld, pnode->location, pexpr->sxBin.pnode1->location,

funcInfo->FindOrAddReferencedPropertyId(propertyId));

break;

}

case knopIndex:

{

EmitBinaryOpnds(pexpr->sxBin.pnode1, pexpr->sxBin.pnode2, byteCodeGenerator, funcInfo);

funcInfo->ReleaseLoc(pexpr->sxBin.pnode2);

funcInfo->ReleaseLoc(pexpr->sxBin.pnode1);

funcInfo->AcquireLoc(pnode);

byteCodeGenerator->Writer()->Element(Js::OpCode::DeleteElemI\_A, pnode->location, pexpr->sxBin.pnode1->location, pexpr->sxBin.pnode2->location);

break;

}

case knopThis:

{

funcInfo->AcquireLoc(pnode);

byteCodeGenerator->Writer()->Reg1(Js::OpCode::LdTrue, pnode->location);

break;

}

default:

{

Emit(pexpr, byteCodeGenerator, funcInfo, false);

funcInfo->ReleaseLoc(pexpr);

byteCodeGenerator->Writer()->Reg2(

Js::OpCode::Delete\_A, funcInfo->AcquireLoc(pnode), pexpr->location);

break;

}

}

byteCodeGenerator->EndStatement(pnode);

break;

}

case knopCall:

{

byteCodeGenerator->StartStatement(pnode);

if (pnode->sxCall.pnodeTarget->nop == knopSuper)

{

byteCodeGenerator->EmitSuperCall(funcInfo, pnode, fReturnValue);

}

else

{

if (pnode->sxCall.isApplyCall && funcInfo->GetApplyEnclosesArgs())

{

// TODO[ianhall]: Can we remove the ApplyCall bytecode gen time optimization?

EmitApplyCall(pnode, Js::Constants::NoRegister, byteCodeGenerator, funcInfo, fReturnValue, true);

}

else

{

EmitCall(pnode, Js::Constants::NoRegister, byteCodeGenerator, funcInfo, fReturnValue, /\*fEvaluateComponents\*/ true, /\*fHasNewTarget\*/ false);

}

}

byteCodeGenerator->EndStatement(pnode);

break;

}

case knopIndex:

{

byteCodeGenerator->StartStatement(pnode);

EmitBinaryOpnds(pnode->sxBin.pnode1, pnode->sxBin.pnode2, byteCodeGenerator, funcInfo);

funcInfo->ReleaseLoc(pnode->sxBin.pnode2);

funcInfo->ReleaseLoc(pnode->sxBin.pnode1);

funcInfo->AcquireLoc(pnode);

Js::RegSlot callObjLocation = pnode->sxBin.pnode1->location;

Js::RegSlot protoLocation = callObjLocation;

EmitSuperMethodBegin(pnode, byteCodeGenerator, funcInfo);

byteCodeGenerator->Writer()->Element(

Js::OpCode::LdElemI\_A, pnode->location, protoLocation, pnode->sxBin.pnode2->location);

byteCodeGenerator->EndStatement(pnode);

break;

}

// this is MemberExpression as rvalue

case knopDot:

{

Emit(pnode->sxBin.pnode1, byteCodeGenerator, funcInfo, false);

funcInfo->ReleaseLoc(pnode->sxBin.pnode1);

funcInfo->AcquireLoc(pnode);

Js::PropertyId propertyId = pnode->sxBin.pnode2->sxPid.PropertyIdFromNameNode();

Js::RegSlot callObjLocation = pnode->sxBin.pnode1->location;

Js::RegSlot protoLocation = callObjLocation;

EmitSuperMethodBegin(pnode, byteCodeGenerator, funcInfo);

if (propertyId == Js::PropertyIds::length)

{

byteCodeGenerator->Writer()->Reg2(Js::OpCode::LdLen\_A, pnode->location, protoLocation);

}

else

{

uint cacheId = funcInfo->FindOrAddInlineCacheId(callObjLocation, propertyId, false, false);

if (pnode->IsCallApplyTargetLoad())

{

byteCodeGenerator->Writer()->PatchableProperty(Js::OpCode::LdFldForCallApplyTarget, pnode->location, protoLocation, cacheId);

}

else

{

if (pnode->sxBin.pnode1->nop == knopSuper)

{

byteCodeGenerator->Writer()->PatchablePropertyWithThisPtr(Js::OpCode::LdSuperFld, pnode->location, protoLocation, funcInfo->thisPointerRegister, cacheId, isConstructorCall);

}

else

{

byteCodeGenerator->Writer()->PatchableProperty(Js::OpCode::LdFld, pnode->location, callObjLocation, cacheId, isConstructorCall);

}

}

}

break;

}

// PTNODE(knopAsg , "=" ,None ,Bin ,fnopBin|fnopAsg)

case knopAsg:

{

ParseNode \*lhs = pnode->sxBin.pnode1;

ParseNode \*rhs = pnode->sxBin.pnode2;

byteCodeGenerator->StartStatement(pnode);

if (pnode->isUsed || fReturnValue)

{

// If the assignment result is used, grab a register to hold it and pass it to EmitAssignment,

// which will copy the assigned value there.

funcInfo->AcquireLoc(pnode);

EmitBinaryReference(lhs, rhs, byteCodeGenerator, funcInfo, false);

EmitAssignment(pnode, lhs, rhs->location, byteCodeGenerator, funcInfo);

}

else

{

EmitBinaryReference(lhs, rhs, byteCodeGenerator, funcInfo, false);

EmitAssignment(nullptr, lhs, rhs->location, byteCodeGenerator, funcInfo);

}

funcInfo->ReleaseLoc(rhs);

if (!(byteCodeGenerator->IsES6DestructuringEnabled() && (lhs->IsPattern())))

{

funcInfo->ReleaseReference(lhs);

}

byteCodeGenerator->EndStatement(pnode);

break;

}

case knopName:

funcInfo->AcquireLoc(pnode);

byteCodeGenerator->EmitPropLoad(pnode->location, pnode->sxPid.sym, pnode->sxPid.pid, funcInfo);

break;

case knopComma:

{

byteCodeGenerator->StartStatement(pnode);

// The parser marks binary opnd pnodes as used, but value of the first opnd of a comma is not used.

// Easier to correct this here than to check every binary op in the parser.

ParseNode \*pnode1 = pnode->sxBin.pnode1;

pnode1->isUsed = false;

if (pnode1->nop == knopComma)

{

// Spot fix for giant comma expressions that send us into OOS if we use a simple recursive

// algorithm. Instead of recursing on comma LHS's, iterate over them, pushing the RHS's onto

// a stack. (This suggests a model for removing recursion from Emit altogether...)

ArenaAllocator \*alloc = byteCodeGenerator->GetAllocator();

SList<ParseNode \*> rhsStack(alloc);

do

{

rhsStack.Push(pnode1->sxBin.pnode2);

pnode1 = pnode1->sxBin.pnode1;

pnode1->isUsed = false;

}

while (pnode1->nop == knopComma);

Emit(pnode1, byteCodeGenerator, funcInfo, false);

if (funcInfo->IsTmpReg(pnode1->location))

{

byteCodeGenerator->Writer()->Reg1(Js::OpCode::Unused, pnode1->location);

}

while (!rhsStack.Empty())

{

ParseNode \*pnodeRhs = rhsStack.Pop();

pnodeRhs->isUsed = false;

Emit(pnodeRhs, byteCodeGenerator, funcInfo, false);

if (funcInfo->IsTmpReg(pnodeRhs->location))

{

byteCodeGenerator->Writer()->Reg1(Js::OpCode::Unused, pnodeRhs->location);

}

funcInfo->ReleaseLoc(pnodeRhs);

}

}

else

{

Emit(pnode1, byteCodeGenerator, funcInfo, false);

if (funcInfo->IsTmpReg(pnode1->location))

{

byteCodeGenerator->Writer()->Reg1(Js::OpCode::Unused, pnode1->location);

}

}

funcInfo->ReleaseLoc(pnode1);

pnode->sxBin.pnode2->isUsed = pnode->isUsed || fReturnValue;

Emit(pnode->sxBin.pnode2, byteCodeGenerator, funcInfo, false);

funcInfo->ReleaseLoc(pnode->sxBin.pnode2);

funcInfo->AcquireLoc(pnode);

if (pnode->sxBin.pnode2->isUsed && pnode->location != pnode->sxBin.pnode2->location)

{

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, pnode->location, pnode->sxBin.pnode2->location);

}

byteCodeGenerator->EndStatement(pnode);

}

break;

// The binary logical ops && and || resolve to the value of the left-hand expression if its

// boolean value short-circuits the operation, and to the value of the right-hand expression

// otherwise. (In other words, the "truth" of the right-hand expression is never tested.)

// PTNODE(knopLogOr , "||" ,None ,Bin ,fnopBin)

case knopLogOr:

{

Js::ByteCodeLabel doneLabel = byteCodeGenerator->Writer()->DefineLabel();

// For boolean expressions that compute a result, we have to burn a register for the result

// so that the back end can identify it cheaply as a single temp lifetime. Revisit this if we do

// full-on renaming in the back end.

funcInfo->AcquireLoc(pnode);

Emit(pnode->sxBin.pnode1, byteCodeGenerator, funcInfo, false);

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, pnode->location, pnode->sxBin.pnode1->location);

byteCodeGenerator->Writer()->BrReg1(Js::OpCode::BrTrue\_A, doneLabel, pnode->sxBin.pnode1->location);

funcInfo->ReleaseLoc(pnode->sxBin.pnode1);

Emit(pnode->sxBin.pnode2, byteCodeGenerator, funcInfo, false);

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, pnode->location, pnode->sxBin.pnode2->location);

funcInfo->ReleaseLoc(pnode->sxBin.pnode2);

byteCodeGenerator->Writer()->MarkLabel(doneLabel);

break;

}

// PTNODE(knopLogAnd , "&&" ,None ,Bin ,fnopBin)

case knopLogAnd:

{

Js::ByteCodeLabel doneLabel = byteCodeGenerator->Writer()->DefineLabel();

// For boolean expressions that compute a result, we have to burn a register for the result

// so that the back end can identify it cheaply as a single temp lifetime. Revisit this if we do

// full-on renaming in the back end.

funcInfo->AcquireLoc(pnode);

Emit(pnode->sxBin.pnode1, byteCodeGenerator, funcInfo, false);

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, pnode->location, pnode->sxBin.pnode1->location);

byteCodeGenerator->Writer()->BrReg1(Js::OpCode::BrFalse\_A, doneLabel, pnode->sxBin.pnode1->location);

funcInfo->ReleaseLoc(pnode->sxBin.pnode1);

Emit(pnode->sxBin.pnode2, byteCodeGenerator, funcInfo, false);

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, pnode->location, pnode->sxBin.pnode2->location);

funcInfo->ReleaseLoc(pnode->sxBin.pnode2);

byteCodeGenerator->Writer()->MarkLabel(doneLabel);

break;

}

// PTNODE(knopQmark , "?" ,None ,Tri ,fnopBin)

case knopQmark:

{

Js::ByteCodeLabel trueLabel = byteCodeGenerator->Writer()->DefineLabel();

Js::ByteCodeLabel falseLabel = byteCodeGenerator->Writer()->DefineLabel();

Js::ByteCodeLabel skipLabel = byteCodeGenerator->Writer()->DefineLabel();

EmitBooleanExpression(pnode->sxTri.pnode1, trueLabel, falseLabel, byteCodeGenerator, funcInfo);

byteCodeGenerator->Writer()->MarkLabel(trueLabel);

funcInfo->ReleaseLoc(pnode->sxTri.pnode1);

// For boolean expressions that compute a result, we have to burn a register for the result

// so that the back end can identify it cheaply as a single temp lifetime. Revisit this if we do

// full-on renaming in the back end.

funcInfo->AcquireLoc(pnode);

Emit(pnode->sxTri.pnode2, byteCodeGenerator, funcInfo, false);

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, pnode->location, pnode->sxTri.pnode2->location);

funcInfo->ReleaseLoc(pnode->sxTri.pnode2);

// Record the branch bytecode offset

byteCodeGenerator->Writer()->RecordStatementAdjustment(Js::FunctionBody::SAT\_FromCurrentToNext);

byteCodeGenerator->Writer()->Br(skipLabel);

byteCodeGenerator->Writer()->MarkLabel(falseLabel);

Emit(pnode->sxTri.pnode3, byteCodeGenerator, funcInfo, false);

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, pnode->location, pnode->sxTri.pnode3->location);

funcInfo->ReleaseLoc(pnode->sxTri.pnode3);

byteCodeGenerator->Writer()->MarkLabel(skipLabel);

break;

}

case knopAsgAdd:

case knopAsgSub:

case knopAsgMul:

case knopAsgDiv:

case knopAsgExpo:

case knopAsgMod:

case knopAsgAnd:

case knopAsgXor:

case knopAsgOr:

case knopAsgLsh:

case knopAsgRsh:

case knopAsgRs2:

byteCodeGenerator->StartStatement(pnode);

// Assign a register for the result only if the result is used or the LHS can't be assigned to

// (i.e., is a constant).

if (pnode->isUsed || fReturnValue || funcInfo->RegIsConst(pnode->sxBin.pnode1->location))

{

// If the assign-op result is used, grab a register to hold it.

funcInfo->AcquireLoc(pnode);

// Grab a register for the initial value and load it.

EmitBinaryReference(pnode->sxBin.pnode1, pnode->sxBin.pnode2, byteCodeGenerator, funcInfo, true);

funcInfo->ReleaseLoc(pnode->sxBin.pnode2);

// Do the arithmetic, store the result, and release the l-value.

byteCodeGenerator->Writer()->Reg3(nopToOp[pnode->nop], pnode->location, pnode->sxBin.pnode1->location,

pnode->sxBin.pnode2->location);

EmitAssignment(pnode, pnode->sxBin.pnode1, pnode->location, byteCodeGenerator, funcInfo);

}

else

{

// Grab a register for the initial value and load it.

EmitBinaryReference(pnode->sxBin.pnode1, pnode->sxBin.pnode2, byteCodeGenerator, funcInfo, true);

funcInfo->ReleaseLoc(pnode->sxBin.pnode2);

// Do the arithmetic, store the result, and release the l-value.

byteCodeGenerator->Writer()->Reg3(nopToOp[pnode->nop], pnode->sxBin.pnode1->location, pnode->sxBin.pnode1->location,

pnode->sxBin.pnode2->location);

EmitAssignment(nullptr, pnode->sxBin.pnode1, pnode->sxBin.pnode1->location, byteCodeGenerator, funcInfo);

}

funcInfo->ReleaseLoad(pnode->sxBin.pnode1);

byteCodeGenerator->EndStatement(pnode);

break;

// General nodes.

// PTNODE(knopTempRef , "temp ref" ,None ,Uni ,fnopUni)

case knopTempRef:

// TODO: check whether mov is necessary

funcInfo->AcquireLoc(pnode);

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, pnode->location, pnode->sxUni.pnode1->location);

break;

// PTNODE(knopTemp , "temp" ,None ,None ,fnopLeaf)

case knopTemp:

// Emit initialization code

if (pnode->sxVar.pnodeInit != nullptr)

{

byteCodeGenerator->StartStatement(pnode);

Emit(pnode->sxVar.pnodeInit, byteCodeGenerator, funcInfo, false);

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, pnode->location, pnode->sxVar.pnodeInit->location);

funcInfo->ReleaseLoc(pnode->sxVar.pnodeInit);

byteCodeGenerator->EndStatement(pnode);

}

break;

// PTNODE(knopVarDecl , "varDcl" ,None ,Var ,fnopNone)

case knopVarDecl:

{

// Emit initialization code

ParseNodePtr initNode = pnode->sxVar.pnodeInit;

if (initNode != nullptr)

{

byteCodeGenerator->StartStatement(pnode);

Emit(pnode->sxVar.pnodeInit, byteCodeGenerator, funcInfo, false);

EmitAssignment(nullptr, pnode, pnode->sxVar.pnodeInit->location, byteCodeGenerator, funcInfo);

funcInfo->ReleaseLoc(pnode->sxVar.pnodeInit);

byteCodeGenerator->EndStatement(pnode);

Symbol \*sym = pnode->sxVar.sym;

if (pnode->sxVar.pnodeInit->nop == knopObject && (ParseNode::Grfnop(pnode->sxVar.pnodeInit->nop) & fnopUni))

{

TrackMemberNodesInObjectForIntConstants(byteCodeGenerator, pnode->sxVar.pnodeInit);

}

else if (initNode->nop == knopInt)

{

TrackIntConstantsOnGlobalObject(byteCodeGenerator, sym);

}

}

break;

}

case knopConstDecl:

{

ParseNodePtr initNode = pnode->sxVar.pnodeInit;

Symbol \* sym = pnode->sxVar.sym;

byteCodeGenerator->StartStatement(pnode);

if (initNode)

{

Emit(initNode, byteCodeGenerator, funcInfo, false);

byteCodeGenerator->EmitPropStore(initNode->location, sym, nullptr, funcInfo, false, true);

funcInfo->ReleaseLoc(pnode->sxVar.pnodeInit);

if (initNode->nop == knopInt)

{

TrackIntConstantsOnGlobalObject(byteCodeGenerator, sym);

}

}

byteCodeGenerator->EndStatement(pnode);

// Set NeedDeclaration to false AFTER emitting the initializer, otherwise we won't have "Use Before Declaration"

// errors reported in the initializer for code like this:

// { const a = a; }

sym->SetNeedDeclaration(false);

break;

}

case knopLetDecl:

{

ParseNodePtr initNode = pnode->sxVar.pnodeInit;

byteCodeGenerator->StartStatement(pnode);

Symbol \* sym = pnode->sxVar.sym;

Js::RegSlot rhsLocation;

if (pnode->sxVar.pnodeInit != nullptr)

{

Emit(pnode->sxVar.pnodeInit, byteCodeGenerator, funcInfo, false);

rhsLocation = pnode->sxVar.pnodeInit->location;

if (initNode->nop == knopInt)

{

TrackIntConstantsOnGlobalObject(byteCodeGenerator, sym);

}

}

else

{

rhsLocation = funcInfo->AcquireTmpRegister();

byteCodeGenerator->Writer()->Reg1(Js::OpCode::LdUndef, rhsLocation);

}

// Set NeedDeclaration to false AFTER emitting the initializer, otherwise we won't have "Use Before Declaration"

// errors reported in the initializer for code like this:

// { let a = a; }

sym->SetNeedDeclaration(false);

byteCodeGenerator->EmitPropStore(rhsLocation, sym, nullptr, funcInfo, true, false);

funcInfo->ReleaseTmpRegister(rhsLocation);

byteCodeGenerator->EndStatement(pnode);

break;

}

// PTNODE(knopFncDecl , "fncDcl" ,None ,Fnc ,fnopLeaf)

case knopFncDecl:

// The "function declarations" were emitted in DefineFunctions()

if (!pnode->sxFnc.IsDeclaration())

{

byteCodeGenerator->DefineOneFunction(pnode, funcInfo, false);

}

break;

// PTNODE(knopClassDecl, "class" ,None ,None ,fnopLeaf)

case knopClassDecl:

{

funcInfo->AcquireLoc(pnode);

// Extends

if (pnode->sxClass.pnodeExtends)

{

Emit(pnode->sxClass.pnodeExtends, byteCodeGenerator, funcInfo, false);

}

Assert(pnode->sxClass.pnodeConstructor);

pnode->sxClass.pnodeConstructor->location = pnode->location;

BeginEmitBlock(pnode->sxClass.pnodeBlock, byteCodeGenerator, funcInfo);

// Constructor

Emit(pnode->sxClass.pnodeConstructor, byteCodeGenerator, funcInfo, false);

EmitComputedFunctionNameVar(bindPnode, pnode->sxClass.pnodeConstructor, byteCodeGenerator);

if (pnode->sxClass.pnodeExtends)

{

byteCodeGenerator->Writer()->InitClass(pnode->location, pnode->sxClass.pnodeExtends->location);

}

else

{

byteCodeGenerator->Writer()->InitClass(pnode->location);

}

Js::RegSlot protoLoc = funcInfo->AcquireTmpRegister(); //register set if we have Instance Methods

int cacheId = funcInfo->FindOrAddInlineCacheId(pnode->location, Js::PropertyIds::prototype, false, false);

byteCodeGenerator->Writer()->PatchableProperty(Js::OpCode::LdFld, protoLoc, pnode->location, cacheId);

byteCodeGenerator->Writer()->Reg2(Js::OpCode::SetHomeObj, pnode->location, protoLoc);

// Static Methods

EmitClassInitializers(pnode->sxClass.pnodeStaticMembers, pnode->location, byteCodeGenerator, funcInfo, pnode, /\*isObjectEmpty\*/ false);

// Instance Methods

EmitClassInitializers(pnode->sxClass.pnodeMembers, protoLoc, byteCodeGenerator, funcInfo, pnode, /\*isObjectEmpty\*/ true);

funcInfo->ReleaseTmpRegister(protoLoc);

// Emit name binding.

if (pnode->sxClass.pnodeName)

{

Symbol \* sym = pnode->sxClass.pnodeName->sxVar.sym;

sym->SetNeedDeclaration(false);

byteCodeGenerator->EmitPropStore(pnode->location, sym, nullptr, funcInfo, false, true);

}

EndEmitBlock(pnode->sxClass.pnodeBlock, byteCodeGenerator, funcInfo);

if (pnode->sxClass.pnodeExtends)

{

funcInfo->ReleaseLoc(pnode->sxClass.pnodeExtends);

}

if (pnode->sxClass.pnodeDeclName)

{

Symbol \* sym = pnode->sxClass.pnodeDeclName->sxVar.sym;

sym->SetNeedDeclaration(false);

byteCodeGenerator->EmitPropStore(pnode->location, sym, nullptr, funcInfo, true, false);

}

break;

}

case knopStrTemplate:

EmitStringTemplate(pnode, byteCodeGenerator, funcInfo);

break;

case knopEndCode:

byteCodeGenerator->Writer()->RecordStatementAdjustment(Js::FunctionBody::SAT\_All);

// load undefined for the fallthrough case:

if (!funcInfo->IsGlobalFunction())

{

if (funcInfo->IsClassConstructor())

{

// For class constructors, we need to explicitly load 'this' into the return register.

byteCodeGenerator->EmitClassConstructorEndCode(funcInfo);

}

else

{

// In the global function, implicit return values are copied to the return register, and if

// necessary the return register is initialized at the top. Don't clobber the value here.

byteCodeGenerator->Writer()->Reg1(Js::OpCode::LdUndef, ByteCodeGenerator::ReturnRegister);

}

}

// Label for non-fall-through return

byteCodeGenerator->Writer()->MarkLabel(funcInfo->singleExit);

if (funcInfo->GetHasCachedScope())

{

byteCodeGenerator->Writer()->AuxNoReg(

Js::OpCode::CommitScope,

funcInfo->localPropIdOffset,

sizeof(Js::PropertyIdArray) + ((funcInfo->GetBodyScope()->GetScopeSlotCount() + 3) \* sizeof(Js::PropertyId)));

}

byteCodeGenerator->StartStatement(pnode);

byteCodeGenerator->Writer()->Empty(Js::OpCode::Ret);

byteCodeGenerator->EndStatement(pnode);

break;

// PTNODE(knopDebugger , "debugger" ,None ,None ,fnopNone)

case knopDebugger:

byteCodeGenerator->StartStatement(pnode);

byteCodeGenerator->Writer()->Empty(Js::OpCode::Break);

byteCodeGenerator->EndStatement(pnode);

break;

// PTNODE(knopFor , "for" ,None ,For ,fnopBreak|fnopContinue)

case knopFor:

if (pnode->sxFor.pnodeInverted != nullptr)

{

byteCodeGenerator->EmitInvertedLoop(pnode, pnode->sxFor.pnodeInverted, funcInfo);

}

else

{

BeginEmitBlock(pnode->sxFor.pnodeBlock, byteCodeGenerator, funcInfo);

Emit(pnode->sxFor.pnodeInit, byteCodeGenerator, funcInfo, false);

funcInfo->ReleaseLoc(pnode->sxFor.pnodeInit);

if (byteCodeGenerator->IsES6ForLoopSemanticsEnabled())

{

CloneEmitBlock(pnode->sxFor.pnodeBlock, byteCodeGenerator, funcInfo);

}

EmitLoop(pnode,

pnode->sxFor.pnodeCond,

pnode->sxFor.pnodeBody,

pnode->sxFor.pnodeIncr,

byteCodeGenerator,

funcInfo,

fReturnValue,

FALSE,

pnode->sxFor.pnodeBlock);

EndEmitBlock(pnode->sxFor.pnodeBlock, byteCodeGenerator, funcInfo);

}

break;

// PTNODE(knopIf , "if" ,None ,If ,fnopNone)

case knopIf:

{

byteCodeGenerator->StartStatement(pnode);

Js::ByteCodeLabel trueLabel = byteCodeGenerator->Writer()->DefineLabel();

Js::ByteCodeLabel falseLabel = byteCodeGenerator->Writer()->DefineLabel();

EmitBooleanExpression(pnode->sxIf.pnodeCond, trueLabel, falseLabel, byteCodeGenerator, funcInfo);

funcInfo->ReleaseLoc(pnode->sxIf.pnodeCond);

byteCodeGenerator->EndStatement(pnode);

byteCodeGenerator->Writer()->MarkLabel(trueLabel);

Emit(pnode->sxIf.pnodeTrue, byteCodeGenerator, funcInfo, fReturnValue);

funcInfo->ReleaseLoc(pnode->sxIf.pnodeTrue);

if (pnode->sxIf.pnodeFalse != nullptr)

{

// has else clause

Js::ByteCodeLabel skipLabel = byteCodeGenerator->Writer()->DefineLabel();

// Record the branch bytecode offset

byteCodeGenerator->Writer()->RecordStatementAdjustment(Js::FunctionBody::SAT\_FromCurrentToNext);

// then clause skips else clause

byteCodeGenerator->Writer()->Br(skipLabel);

// generate code for else clause

byteCodeGenerator->Writer()->MarkLabel(falseLabel);

Emit(pnode->sxIf.pnodeFalse, byteCodeGenerator, funcInfo, fReturnValue);

funcInfo->ReleaseLoc(pnode->sxIf.pnodeFalse);

byteCodeGenerator->Writer()->MarkLabel(skipLabel);

}

else

{

byteCodeGenerator->Writer()->MarkLabel(falseLabel);

}

if (pnode->emitLabels)

{

byteCodeGenerator->Writer()->MarkLabel(pnode->sxStmt.breakLabel);

}

break;

}

case knopWhile:

EmitLoop(pnode,

pnode->sxWhile.pnodeCond,

pnode->sxWhile.pnodeBody,

nullptr,

byteCodeGenerator,

funcInfo,

fReturnValue);

break;

// PTNODE(knopDoWhile , "do-while" ,None ,While,fnopBreak|fnopContinue)

case knopDoWhile:

EmitLoop(pnode,

pnode->sxWhile.pnodeCond,

pnode->sxWhile.pnodeBody,

nullptr,

byteCodeGenerator,

funcInfo,

fReturnValue,

true);

break;

// PTNODE(knopForIn , "for in" ,None ,ForIn,fnopBreak|fnopContinue|fnopCleanup)

case knopForIn:

EmitForInOrForOf(pnode, byteCodeGenerator, funcInfo, fReturnValue);

break;

case knopForOf:

EmitForInOrForOf(pnode, byteCodeGenerator, funcInfo, fReturnValue);

break;

// PTNODE(knopReturn , "return" ,None ,Uni ,fnopNone)

case knopReturn:

byteCodeGenerator->StartStatement(pnode);

if (pnode->sxReturn.pnodeExpr != nullptr)

{

if (pnode->sxReturn.pnodeExpr->location == Js::Constants::NoRegister)

{

// No need to burn a register for the return value. If we need a temp, use R0 directly.

pnode->sxReturn.pnodeExpr->location = ByteCodeGenerator::ReturnRegister;

}

Emit(pnode->sxReturn.pnodeExpr, byteCodeGenerator, funcInfo, fReturnValue);

if (pnode->sxReturn.pnodeExpr->location != ByteCodeGenerator::ReturnRegister)

{

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, ByteCodeGenerator::ReturnRegister, pnode->sxReturn.pnodeExpr->location);

}

funcInfo->GetParsedFunctionBody()->SetHasNoExplicitReturnValue(false);

}

else

{

byteCodeGenerator->Writer()->Reg1(Js::OpCode::LdUndef, ByteCodeGenerator::ReturnRegister);

}

if (funcInfo->IsClassConstructor())

{

// return expr; // becomes like below:

//

// if (IsObject(expr)) {

// return expr;

// } else if (IsBaseClassConstructor) {

// return this;

// } else if (!IsUndefined(expr)) {

// throw TypeError;

// }

Js::ByteCodeLabel returnExprLabel = byteCodeGenerator->Writer()->DefineLabel();

byteCodeGenerator->Writer()->BrReg1(Js::OpCode::BrOnObject\_A, returnExprLabel, ByteCodeGenerator::ReturnRegister);

if (funcInfo->IsBaseClassConstructor())

{

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, ByteCodeGenerator::ReturnRegister, funcInfo->thisPointerRegister);

}

else

{

Js::ByteCodeLabel returnThisLabel = byteCodeGenerator->Writer()->DefineLabel();

byteCodeGenerator->Writer()->BrReg2(Js::OpCode::BrSrEq\_A, returnThisLabel, ByteCodeGenerator::ReturnRegister, funcInfo->undefinedConstantRegister);

byteCodeGenerator->Writer()->W1(Js::OpCode::RuntimeTypeError, SCODE\_CODE(JSERR\_ClassDerivedConstructorInvalidReturnType));

byteCodeGenerator->Writer()->MarkLabel(returnThisLabel);

byteCodeGenerator->EmitClassConstructorEndCode(funcInfo);

}

byteCodeGenerator->Writer()->MarkLabel(returnExprLabel);

}

if (pnode->sxStmt.grfnop & fnopCleanup)

{

EmitJumpCleanup(pnode, nullptr, byteCodeGenerator, funcInfo);

}

byteCodeGenerator->Writer()->Br(funcInfo->singleExit);

byteCodeGenerator->EndStatement(pnode);

break;

case knopLabel:

break;

// PTNODE(knopBlock , "{}" ,None ,Block,fnopNone)

case knopBlock:

if (pnode->sxBlock.pnodeStmt != nullptr)

{

EmitBlock(pnode, byteCodeGenerator, funcInfo, fReturnValue);

if (pnode->emitLabels)

{

byteCodeGenerator->Writer()->MarkLabel(pnode->sxStmt.breakLabel);

}

}

break;

// PTNODE(knopWith , "with" ,None ,With ,fnopCleanup)

case knopWith:

{

Assert(pnode->sxWith.pnodeObj != nullptr);

byteCodeGenerator->StartStatement(pnode);

// Copy the with object to a temp register (the location assigned to pnode) so that if the with object

// is overwritten in the body, the lookups are not affected.

funcInfo->AcquireLoc(pnode);

Emit(pnode->sxWith.pnodeObj, byteCodeGenerator, funcInfo, false);

Js::RegSlot regVal = (byteCodeGenerator->GetScriptContext()->GetConfig()->IsES6UnscopablesEnabled()) ? funcInfo->AcquireTmpRegister() : pnode->location;

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Conv\_Obj, regVal, pnode->sxWith.pnodeObj->location);

if (byteCodeGenerator->GetScriptContext()->GetConfig()->IsES6UnscopablesEnabled())

{

byteCodeGenerator->Writer()->Reg2(Js::OpCode::NewWithObject, pnode->location, regVal);

}

byteCodeGenerator->EndStatement(pnode);

#ifdef PERF\_HINT

if (PHASE\_TRACE1(Js::PerfHintPhase))

{

WritePerfHint(PerfHints::HasWithBlock, funcInfo->byteCodeFunction->GetFunctionBody(), byteCodeGenerator->Writer()->GetCurrentOffset() - 1);

}

#endif

if (pnode->sxWith.pnodeBody != nullptr)

{

Scope \*scope = pnode->sxWith.scope;

scope->SetLocation(pnode->location);

byteCodeGenerator->PushScope(scope);

Js::DebuggerScope \*debuggerScope = byteCodeGenerator->RecordStartScopeObject(pnode, Js::DiagExtraScopesType::DiagWithScope, regVal);

if (byteCodeGenerator->ShouldTrackDebuggerMetadata())

{

byteCodeGenerator->Writer()->AddPropertyToDebuggerScope(debuggerScope, regVal, Js::Constants::NoProperty, /\*shouldConsumeRegister\*/ true, Js::DebuggerScopePropertyFlags\_WithObject);

}

Emit(pnode->sxWith.pnodeBody, byteCodeGenerator, funcInfo, fReturnValue);

funcInfo->ReleaseLoc(pnode->sxWith.pnodeBody);

byteCodeGenerator->PopScope();

byteCodeGenerator->RecordEndScopeObject(pnode);

}

if (pnode->emitLabels)

{

byteCodeGenerator->Writer()->MarkLabel(pnode->sxStmt.breakLabel);

}

if (byteCodeGenerator->GetScriptContext()->GetConfig()->IsES6UnscopablesEnabled())

{

funcInfo->ReleaseTmpRegister(regVal);

}

funcInfo->ReleaseLoc(pnode->sxWith.pnodeObj);

break;

}

// PTNODE(knopBreak , "break" ,None ,Jump ,fnopNone)

case knopBreak:

Assert(pnode->sxJump.pnodeTarget->emitLabels);

byteCodeGenerator->StartStatement(pnode);

if (pnode->sxStmt.grfnop & fnopCleanup)

{

EmitJumpCleanup(pnode, pnode->sxJump.pnodeTarget, byteCodeGenerator, funcInfo);

}

byteCodeGenerator->Writer()->Br(pnode->sxJump.pnodeTarget->sxStmt.breakLabel);

if (pnode->emitLabels)

{

byteCodeGenerator->Writer()->MarkLabel(pnode->sxStmt.breakLabel);

}

byteCodeGenerator->EndStatement(pnode);

break;

case knopContinue:

Assert(pnode->sxJump.pnodeTarget->emitLabels);

byteCodeGenerator->StartStatement(pnode);

if (pnode->sxStmt.grfnop & fnopCleanup)

{

EmitJumpCleanup(pnode, pnode->sxJump.pnodeTarget, byteCodeGenerator, funcInfo);

}

byteCodeGenerator->Writer()->Br(pnode->sxJump.pnodeTarget->sxStmt.continueLabel);

byteCodeGenerator->EndStatement(pnode);

break;

// PTNODE(knopContinue , "continue" ,None ,Jump ,fnopNone)

case knopSwitch:

{

BOOL fHasDefault = false;

Assert(pnode->sxSwitch.pnodeVal != nullptr);

byteCodeGenerator->StartStatement(pnode);

Emit(pnode->sxSwitch.pnodeVal, byteCodeGenerator, funcInfo, false);

Js::RegSlot regVal = funcInfo->AcquireTmpRegister();

byteCodeGenerator->Writer()->Reg2(Js::OpCode::BeginSwitch, regVal, pnode->sxSwitch.pnodeVal->location);

BeginEmitBlock(pnode->sxSwitch.pnodeBlock, byteCodeGenerator, funcInfo);

byteCodeGenerator->EndStatement(pnode);

// TODO: if all cases are compile-time constants, emit a switch statement in the byte

// code so the BE can optimize it.

ParseNode \*pnodeCase;

for (pnodeCase = pnode->sxSwitch.pnodeCases; pnodeCase; pnodeCase = pnodeCase->sxCase.pnodeNext)

{

// Jump to the first case body if this one doesn't match. Make sure any side-effects of the case

// expression take place regardless.

pnodeCase->sxCase.labelCase = byteCodeGenerator->Writer()->DefineLabel();

if (pnodeCase == pnode->sxSwitch.pnodeDefault)

{

fHasDefault = true;

continue;

}

Emit(pnodeCase->sxCase.pnodeExpr, byteCodeGenerator, funcInfo, false);

byteCodeGenerator->Writer()->BrReg2(

Js::OpCode::Case, pnodeCase->sxCase.labelCase, regVal, pnodeCase->sxCase.pnodeExpr->location);

funcInfo->ReleaseLoc(pnodeCase->sxCase.pnodeExpr);

}

// No explicit case value matches. Jump to the default arm (if any) or break out altogether.

if (fHasDefault)

{

byteCodeGenerator->Writer()->Br(Js::OpCode::EndSwitch, pnode->sxSwitch.pnodeDefault->sxCase.labelCase);

}

else

{

if (!pnode->emitLabels)

{

pnode->sxStmt.breakLabel = byteCodeGenerator->Writer()->DefineLabel();

}

byteCodeGenerator->Writer()->Br(Js::OpCode::EndSwitch, pnode->sxStmt.breakLabel);

}

// Now emit the case arms to which we jump on matching a case value.

for (pnodeCase = pnode->sxSwitch.pnodeCases; pnodeCase; pnodeCase = pnodeCase->sxCase.pnodeNext)

{

byteCodeGenerator->Writer()->MarkLabel(pnodeCase->sxCase.labelCase);

Emit(pnodeCase->sxCase.pnodeBody, byteCodeGenerator, funcInfo, fReturnValue);

funcInfo->ReleaseLoc(pnodeCase->sxCase.pnodeBody);

}

EndEmitBlock(pnode->sxSwitch.pnodeBlock, byteCodeGenerator, funcInfo);

funcInfo->ReleaseTmpRegister(regVal);

funcInfo->ReleaseLoc(pnode->sxSwitch.pnodeVal);

if (!fHasDefault || pnode->emitLabels)

{

byteCodeGenerator->Writer()->MarkLabel(pnode->sxStmt.breakLabel);

}

break;

}

case knopTryCatch:

{

Js::ByteCodeLabel catchLabel = (Js::ByteCodeLabel) - 1;

ParseNode \*pnodeTry = pnode->sxTryCatch.pnodeTry;

Assert(pnodeTry);

ParseNode \*pnodeCatch = pnode->sxTryCatch.pnodeCatch;

Assert(pnodeCatch);

catchLabel = byteCodeGenerator->Writer()->DefineLabel();

// Note: try uses OpCode::Leave which causes a return to parent interpreter thunk,

// same for catch block. Thus record cross interpreter frame entry/exit records for them.

byteCodeGenerator->Writer()->RecordCrossFrameEntryExitRecord(/\* isEnterBlock = \*/ true);

byteCodeGenerator->Writer()->Br(Js::OpCode::TryCatch, catchLabel);

ByteCodeGenerator::TryScopeRecord tryRecForTry(Js::OpCode::TryCatch, catchLabel);

if (funcInfo->byteCodeFunction->IsGenerator())

{

byteCodeGenerator->tryScopeRecordsList.LinkToEnd(&tryRecForTry);

}

Emit(pnodeTry->sxTry.pnodeBody, byteCodeGenerator, funcInfo, fReturnValue);

funcInfo->ReleaseLoc(pnodeTry->sxTry.pnodeBody);

if (funcInfo->byteCodeFunction->IsGenerator())

{

byteCodeGenerator->tryScopeRecordsList.UnlinkFromEnd();

}

byteCodeGenerator->Writer()->RecordCrossFrameEntryExitRecord(/\* isEnterBlock = \*/ false);

byteCodeGenerator->Writer()->Empty(Js::OpCode::Leave);

byteCodeGenerator->Writer()->Br(pnode->sxStmt.breakLabel);

byteCodeGenerator->Writer()->MarkLabel(catchLabel);

Assert(pnodeCatch->sxCatch.pnodeParam);

ParseNode \*pnodeObj = pnodeCatch->sxCatch.pnodeParam;

Js::RegSlot location;

bool acquiredTempLocation = false;

Js::DebuggerScope \*debuggerScope = nullptr;

Js::DebuggerScopePropertyFlags debuggerPropertyFlags = Js::DebuggerScopePropertyFlags\_CatchObject;

bool isPattern = pnodeObj->nop == knopParamPattern;

if (isPattern)

{

location = pnodeObj->sxParamPattern.location;

}

else

{

location = pnodeObj->sxPid.sym->GetLocation();

}

if (location == Js::Constants::NoRegister)

{

location = funcInfo->AcquireLoc(pnodeObj);

acquiredTempLocation = true;

}

byteCodeGenerator->Writer()->Reg1(Js::OpCode::Catch, location);

Scope \*scope = pnodeCatch->sxCatch.scope;

if (isPattern || scope->GetMustInstantiate())

{

byteCodeGenerator->PushScope(scope);

}

if (scope->GetMustInstantiate())

{

Assert(scope->GetLocation() == Js::Constants::NoRegister);

if (scope->GetIsObject())

{

debuggerScope = byteCodeGenerator->RecordStartScopeObject(pnode, Js::DiagCatchScopeInObject, funcInfo->InnerScopeToRegSlot(scope));

byteCodeGenerator->Writer()->Unsigned1(Js::OpCode::NewPseudoScope, scope->GetInnerScopeIndex());

}

else

{

int index = Js::DebuggerScope::InvalidScopeIndex;

debuggerScope = byteCodeGenerator->RecordStartScopeObject(pnode, Js::DiagCatchScopeInSlot, funcInfo->InnerScopeToRegSlot(scope), &index);

byteCodeGenerator->Writer()->Num3(Js::OpCode::NewInnerScopeSlots, scope->GetInnerScopeIndex(), scope->GetScopeSlotCount() + Js::ScopeSlots::FirstSlotIndex, index);

}

}

else

{

debuggerScope = byteCodeGenerator->RecordStartScopeObject(pnode, Js::DiagCatchScopeDirect, location);

}

auto ParamTrackAndInitialization = [&](Symbol \*sym, bool initializeParam, Js::RegSlot location)

{

if (sym->IsInSlot(funcInfo))

{

Assert(scope->GetMustInstantiate());

if (scope->GetIsObject())

{

Js::OpCode op = (sym->GetDecl()->nop == knopLetDecl) ? Js::OpCode::InitUndeclLetFld :

byteCodeGenerator->GetInitFldOp(scope, scope->GetLocation(), funcInfo, false);

Js::PropertyId propertyId = sym->EnsurePosition(byteCodeGenerator);

uint cacheId = funcInfo->FindOrAddInlineCacheId(funcInfo->InnerScopeToRegSlot(scope), propertyId, false, true);

byteCodeGenerator->Writer()->ElementPIndexed(op, location, scope->GetInnerScopeIndex(), cacheId);

byteCodeGenerator->TrackActivationObjectPropertyForDebugger(debuggerScope, sym, debuggerPropertyFlags);

}

else

{

byteCodeGenerator->TrackSlotArrayPropertyForDebugger(debuggerScope, sym, sym->EnsurePosition(byteCodeGenerator), debuggerPropertyFlags);

if (initializeParam)

{

byteCodeGenerator->EmitLocalPropInit(location, sym, funcInfo);

}

else

{

Js::RegSlot tmpReg = funcInfo->AcquireTmpRegister();

byteCodeGenerator->Writer()->Reg1(Js::OpCode::InitUndecl, tmpReg);

byteCodeGenerator->EmitLocalPropInit(tmpReg, sym, funcInfo);

funcInfo->ReleaseTmpRegister(tmpReg);

}

}

}

else

{

byteCodeGenerator->TrackRegisterPropertyForDebugger(debuggerScope, sym, funcInfo, debuggerPropertyFlags);

if (initializeParam)

{

byteCodeGenerator->EmitLocalPropInit(location, sym, funcInfo);

}

else

{

byteCodeGenerator->Writer()->Reg1(Js::OpCode::InitUndecl, location);

}

}

};

if (isPattern)

{

Parser::MapBindIdentifier(pnodeObj->sxParamPattern.pnode1, [&](ParseNodePtr item)

{

ParamTrackAndInitialization(item->sxVar.sym, false /\*initializeParam\*/, item->sxVar.sym->GetLocation());

});

byteCodeGenerator->Writer()->RecordCrossFrameEntryExitRecord(true);

// Now emitting bytecode for destructuring pattern

byteCodeGenerator->StartStatement(pnodeCatch);

ParseNodePtr pnode1 = pnodeObj->sxParamPattern.pnode1;

Assert(pnode1->IsPattern());

EmitAssignment(nullptr, pnode1, location, byteCodeGenerator, funcInfo);

byteCodeGenerator->EndStatement(pnodeCatch);

}

else

{

ParamTrackAndInitialization(pnodeObj->sxPid.sym, true /\*initializeParam\*/, location);

if (scope->GetMustInstantiate())

{

pnodeObj->sxPid.sym->SetIsGlobalCatch(true);

}

byteCodeGenerator->Writer()->RecordCrossFrameEntryExitRecord(true);

// Allow a debugger to stop on the 'catch (e)'

byteCodeGenerator->StartStatement(pnodeCatch);

byteCodeGenerator->Writer()->Empty(Js::OpCode::Nop);

byteCodeGenerator->EndStatement(pnodeCatch);

}

ByteCodeGenerator::TryScopeRecord tryRecForCatch(Js::OpCode::ResumeCatch, catchLabel);

if (funcInfo->byteCodeFunction->IsGenerator())

{

byteCodeGenerator->tryScopeRecordsList.LinkToEnd(&tryRecForCatch);

}

Emit(pnodeCatch->sxCatch.pnodeBody, byteCodeGenerator, funcInfo, fReturnValue);

if (funcInfo->byteCodeFunction->IsGenerator())

{

byteCodeGenerator->tryScopeRecordsList.UnlinkFromEnd();

}

if (scope->GetMustInstantiate() || isPattern)

{

byteCodeGenerator->PopScope();

}

byteCodeGenerator->RecordEndScopeObject(pnode);

funcInfo->ReleaseLoc(pnodeCatch->sxCatch.pnodeBody);

if (acquiredTempLocation)

{

funcInfo->ReleaseLoc(pnodeObj);

}

byteCodeGenerator->Writer()->RecordCrossFrameEntryExitRecord(false);

byteCodeGenerator->Writer()->Empty(Js::OpCode::Leave);

byteCodeGenerator->Writer()->MarkLabel(pnode->sxStmt.breakLabel);

break;

}

case knopTryFinally:

{

Js::ByteCodeLabel finallyLabel = (Js::ByteCodeLabel) - 1;

ParseNode \*pnodeTry = pnode->sxTryFinally.pnodeTry;

Assert(pnodeTry);

ParseNode \*pnodeFinally = pnode->sxTryFinally.pnodeFinally;

Assert(pnodeFinally);

// If we yield from the finally block after an exception, we have to store the exception object for the future next call.

// When we yield from the Try-Finally the offset to the end of the Try block is needed for the branch instruction.

Js::RegSlot regException = Js::Constants::NoRegister;

Js::RegSlot regOffset = Js::Constants::NoRegister;

finallyLabel = byteCodeGenerator->Writer()->DefineLabel();

byteCodeGenerator->Writer()->RecordCrossFrameEntryExitRecord(true);

// [CONSIDER][aneeshd] Ideally the TryFinallyWithYield opcode needs to be used only if there is a yield expression.

// For now, if the function is generator we are using the TryFinallyWithYield.

ByteCodeGenerator::TryScopeRecord tryRecForTry(Js::OpCode::TryFinallyWithYield, finallyLabel);

if (funcInfo->byteCodeFunction->IsGenerator())

{

regException = funcInfo->AcquireTmpRegister();

regOffset = funcInfo->AcquireTmpRegister();

byteCodeGenerator->Writer()->BrReg2(Js::OpCode::TryFinallyWithYield, finallyLabel, regException, regOffset);

tryRecForTry.reg1 = regException;

tryRecForTry.reg2 = regOffset;

byteCodeGenerator->tryScopeRecordsList.LinkToEnd(&tryRecForTry);

}

else

{

byteCodeGenerator->Writer()->Br(Js::OpCode::TryFinally, finallyLabel);

}

Emit(pnodeTry->sxTry.pnodeBody, byteCodeGenerator, funcInfo, fReturnValue);

funcInfo->ReleaseLoc(pnodeTry->sxTry.pnodeBody);

if (funcInfo->byteCodeFunction->IsGenerator())

{

byteCodeGenerator->tryScopeRecordsList.UnlinkFromEnd();

}

byteCodeGenerator->Writer()->Empty(Js::OpCode::Leave);

byteCodeGenerator->Writer()->RecordCrossFrameEntryExitRecord(false);

// Note: although we don't use OpCode::Leave for finally block,

// OpCode::LeaveNull causes a return to parent interpreter thunk.

// This has to be on offset prior to offset of 1st statement of finally.

byteCodeGenerator->Writer()->RecordCrossFrameEntryExitRecord(true);

byteCodeGenerator->Writer()->Br(pnode->sxStmt.breakLabel);

byteCodeGenerator->Writer()->MarkLabel(finallyLabel);

ByteCodeGenerator::TryScopeRecord tryRecForFinally(Js::OpCode::ResumeFinally, finallyLabel, regException, regOffset);

if (funcInfo->byteCodeFunction->IsGenerator())

{

byteCodeGenerator->tryScopeRecordsList.LinkToEnd(&tryRecForFinally);

}

Emit(pnodeFinally->sxFinally.pnodeBody, byteCodeGenerator, funcInfo, fReturnValue);

funcInfo->ReleaseLoc(pnodeFinally->sxFinally.pnodeBody);

if (funcInfo->byteCodeFunction->IsGenerator())

{

byteCodeGenerator->tryScopeRecordsList.UnlinkFromEnd();

funcInfo->ReleaseTmpRegister(regOffset);

funcInfo->ReleaseTmpRegister(regException);

}

byteCodeGenerator->Writer()->RecordCrossFrameEntryExitRecord(false);

byteCodeGenerator->Writer()->Empty(Js::OpCode::LeaveNull);

byteCodeGenerator->Writer()->MarkLabel(pnode->sxStmt.breakLabel);

break;

}

case knopThrow:

byteCodeGenerator->StartStatement(pnode);

Emit(pnode->sxUni.pnode1, byteCodeGenerator, funcInfo, false);

byteCodeGenerator->Writer()->Reg1(Js::OpCode::Throw, pnode->sxUni.pnode1->location);

funcInfo->ReleaseLoc(pnode->sxUni.pnode1);

byteCodeGenerator->EndStatement(pnode);

break;

case knopYieldLeaf:

byteCodeGenerator->StartStatement(pnode);

funcInfo->AcquireLoc(pnode);

EmitYield(funcInfo->undefinedConstantRegister, pnode->location, byteCodeGenerator, funcInfo);

byteCodeGenerator->EndStatement(pnode);

break;

case knopAwait:

case knopYield:

byteCodeGenerator->StartStatement(pnode);

funcInfo->AcquireLoc(pnode);

Emit(pnode->sxUni.pnode1, byteCodeGenerator, funcInfo, false);

EmitYield(pnode->sxUni.pnode1->location, pnode->location, byteCodeGenerator, funcInfo);

funcInfo->ReleaseLoc(pnode->sxUni.pnode1);

byteCodeGenerator->EndStatement(pnode);

break;

case knopYieldStar:

byteCodeGenerator->StartStatement(pnode);

EmitYieldStar(pnode, byteCodeGenerator, funcInfo);

byteCodeGenerator->EndStatement(pnode);

break;

case knopAsyncSpawn:

EmitBinary(Js::OpCode::AsyncSpawn, pnode, byteCodeGenerator, funcInfo);

break;

default:

AssertMsg(0, "emit unhandled pnode op");

break;

}

if (fReturnValue && IsExpressionStatement(pnode, byteCodeGenerator->GetScriptContext()))

{

// If this statement may produce the global function's return value, copy its result to the return register.

// fReturnValue implies global function, which implies that "return" is a parse error.

Assert(funcInfo->IsGlobalFunction());

Assert(pnode->nop != knopReturn);

byteCodeGenerator->Writer()->Reg2(Js::OpCode::Ld\_A, ByteCodeGenerator::ReturnRegister, pnode->location);

}

}

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

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

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

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

#include "RuntimeByteCodePch.h"

#include "FormalsUtil.h"

#include "Library\StackScriptFunction.h"

void PreVisitBlock(ParseNode \*pnodeBlock, ByteCodeGenerator \*byteCodeGenerator);

void PostVisitBlock(ParseNode \*pnodeBlock, ByteCodeGenerator \*byteCodeGenerator);

bool IsCallOfConstants(ParseNode \*pnode)

{

return pnode->sxCall.callOfConstants && pnode->sxCall.argCount > ByteCodeGenerator::MinArgumentsForCallOptimization;

}

template <class PrefixFn, class PostfixFn>

void Visit(ParseNode \*pnode, ByteCodeGenerator\* byteCodeGenerator, PrefixFn prefix, PostfixFn postfix, ParseNode \* pnodeParent = nullptr);

template<class TContext>

void VisitIndirect(ParseNode\* pnode, ByteCodeGenerator\* byteCodeGenerator, TContext\* context,

void (\*prefix)(ParseNode\* pnode, ByteCodeGenerator\* byteCodeGenerator, TContext\* context),

void (\*postfix)(ParseNode\* pnode, ByteCodeGenerator\* byteCodeGenerator, TContext\* context),

ParseNode \*pnodeParent = nullptr)

{

Visit(pnode, byteCodeGenerator,

[context, prefix](ParseNode \* pnode, ByteCodeGenerator \* byteCodeGenerator)

{

prefix(pnode, byteCodeGenerator, context);

},

[context, postfix](ParseNode \* pnode, ByteCodeGenerator \* byteCodeGenerator)

{

if (postfix)

{

postfix(pnode, byteCodeGenerator, context);

}

}, pnodeParent);

}

template <class PrefixFn, class PostfixFn>

void VisitList(ParseNode \*pnode, ByteCodeGenerator\* byteCodeGenerator, PrefixFn prefix, PostfixFn postfix)

{

Assert(pnode != nullptr);

Assert(pnode->nop == knopList);

do

{

ParseNode \* pnode1 = pnode->sxBin.pnode1;

Visit(pnode1, byteCodeGenerator, prefix, postfix);

pnode = pnode->sxBin.pnode2;

}

while (pnode->nop == knopList);

Visit(pnode, byteCodeGenerator, prefix, postfix);

}

template <class PrefixFn, class PostfixFn>

void VisitWithStmt(ParseNode \*pnode, Js::RegSlot loc, ByteCodeGenerator\* byteCodeGenerator, PrefixFn prefix, PostfixFn postfix, ParseNode \*pnodeParent)

{

// Note the fact that we're visiting the body of a with statement. This allows us to optimize register assignment

// in the normal case of calls not requiring that their "this" objects be found dynamically.

Scope \*scope = pnode->sxWith.scope;

byteCodeGenerator->PushScope(scope);

Visit(pnode->sxWith.pnodeBody, byteCodeGenerator, prefix, postfix, pnodeParent);

scope->SetIsObject();

scope->SetMustInstantiate(true);

byteCodeGenerator->PopScope();

}

bool BlockHasOwnScope(ParseNode\* pnodeBlock, ByteCodeGenerator \*byteCodeGenerator)

{

Assert(pnodeBlock->nop == knopBlock);

return pnodeBlock->sxBlock.scope != nullptr &&

(!(pnodeBlock->grfpn & fpnSyntheticNode) ||

(pnodeBlock->sxBlock.blockType == PnodeBlockType::Global && byteCodeGenerator->IsEvalWithBlockScopingNoParentScopeInfo()));

}

void BeginVisitBlock(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator)

{

if (BlockHasOwnScope(pnode, byteCodeGenerator))

{

Scope \*scope = pnode->sxBlock.scope;

FuncInfo \*func = scope->GetFunc();

if (scope->IsInnerScope())

{

// Give this scope an index so its slots can be accessed via the index in the byte code,

// not a register.

scope->SetInnerScopeIndex(func->AcquireInnerScopeIndex());

}

byteCodeGenerator->PushBlock(pnode);

byteCodeGenerator->PushScope(pnode->sxBlock.scope);

}

}

void EndVisitBlock(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator)

{

if (BlockHasOwnScope(pnode, byteCodeGenerator))

{

Scope \*scope = pnode->sxBlock.scope;

FuncInfo \*func = scope->GetFunc();

if (!byteCodeGenerator->IsInDebugMode() &&

scope->HasInnerScopeIndex())

{

// In debug mode, don't release the current index, as we're giving each scope a unique index, regardless

// of nesting.

Assert(scope->GetInnerScopeIndex() == func->CurrentInnerScopeIndex());

func->ReleaseInnerScopeIndex();

}

Assert(byteCodeGenerator->GetCurrentScope() == scope);

byteCodeGenerator->PopScope();

byteCodeGenerator->PopBlock();

}

}

void BeginVisitCatch(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator)

{

Scope \*scope = pnode->sxCatch.scope;

FuncInfo \*func = scope->GetFunc();

if (func->GetCallsEval() || func->GetChildCallsEval() ||

(byteCodeGenerator->GetFlags() & (fscrEval | fscrImplicitThis | fscrImplicitParents)))

{

scope->SetIsObject();

}

// Give this scope an index so its slots can be accessed via the index in the byte code,

// not a register.

scope->SetInnerScopeIndex(func->AcquireInnerScopeIndex());

byteCodeGenerator->PushScope(pnode->sxCatch.scope);

}

void EndVisitCatch(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator)

{

Scope \*scope = pnode->sxCatch.scope;

if (scope->HasInnerScopeIndex() && !byteCodeGenerator->IsInDebugMode())

{

// In debug mode, don't release the current index, as we're giving each scope a unique index,

// regardless of nesting.

FuncInfo \*func = scope->GetFunc();

Assert(scope->GetInnerScopeIndex() == func->CurrentInnerScopeIndex());

func->ReleaseInnerScopeIndex();

}

byteCodeGenerator->PopScope();

}

bool CreateNativeArrays(ByteCodeGenerator \*byteCodeGenerator, FuncInfo \*funcInfo)

{

#if ENABLE\_PROFILE\_INFO

Js::FunctionBody \*functionBody = funcInfo ? funcInfo->GetParsedFunctionBody() : nullptr;

return

!PHASE\_OFF\_OPTFUNC(Js::NativeArrayPhase, functionBody) &&

!byteCodeGenerator->IsInDebugMode() &&

(

functionBody

? Js::DynamicProfileInfo::IsEnabled(Js::NativeArrayPhase, functionBody)

: Js::DynamicProfileInfo::IsEnabledForAtLeastOneFunction(

Js::NativeArrayPhase,

byteCodeGenerator->GetScriptContext())

);

#else

return false;

#endif

}

bool EmitAsConstantArray(ParseNode \*pnodeArr, ByteCodeGenerator \*byteCodeGenerator)

{

Assert(pnodeArr && pnodeArr->nop == knopArray);

// TODO: We shouldn't have to handle an empty funcinfo stack here, but there seem to be issues

// with the stack involved nested deferral. Remove this null check when those are resolved.

if (CreateNativeArrays(byteCodeGenerator, byteCodeGenerator->TopFuncInfo()))

{

return pnodeArr->sxArrLit.arrayOfNumbers;

}

return pnodeArr->sxArrLit.arrayOfTaggedInts && pnodeArr->sxArrLit.count > 1;

}

void PropagateFlags(ParseNode \*pnodeChild, ParseNode \*pnodeParent);

template<class PrefixFn, class PostfixFn>

void Visit(ParseNode \*pnode, ByteCodeGenerator\* byteCodeGenerator, PrefixFn prefix, PostfixFn postfix, ParseNode \*pnodeParent)

{

if (pnode == nullptr)

{

return;

}

ThreadContext::ProbeCurrentStackNoDispose(Js::Constants::MinStackByteCodeVisitor, byteCodeGenerator->GetScriptContext());

prefix(pnode, byteCodeGenerator);

switch (pnode->nop)

{

default:

{

uint flags = ParseNode::Grfnop(pnode->nop);

if (flags&fnopUni)

{

Visit(pnode->sxUni.pnode1, byteCodeGenerator, prefix, postfix);

}

else if (flags&fnopBin)

{

Visit(pnode->sxBin.pnode1, byteCodeGenerator, prefix, postfix);

Visit(pnode->sxBin.pnode2, byteCodeGenerator, prefix, postfix);

}

break;

}

case knopParamPattern:

Visit(pnode->sxParamPattern.pnode1, byteCodeGenerator, prefix, postfix);

break;

case knopArrayPattern:

if (!byteCodeGenerator->InDestructuredPattern())

{

byteCodeGenerator->SetInDestructuredPattern(true);

Visit(pnode->sxUni.pnode1, byteCodeGenerator, prefix, postfix);

byteCodeGenerator->SetInDestructuredPattern(false);

}

else

{

Visit(pnode->sxUni.pnode1, byteCodeGenerator, prefix, postfix);

}

break;

case knopCall:

Visit(pnode->sxCall.pnodeTarget, byteCodeGenerator, prefix, postfix);

Visit(pnode->sxCall.pnodeArgs, byteCodeGenerator, prefix, postfix);

break;

case knopNew:

{

Visit(pnode->sxCall.pnodeTarget, byteCodeGenerator, prefix, postfix);

if (!IsCallOfConstants(pnode))

{

Visit(pnode->sxCall.pnodeArgs, byteCodeGenerator, prefix, postfix);

}

break;

}

case knopQmark:

Visit(pnode->sxTri.pnode1, byteCodeGenerator, prefix, postfix);

Visit(pnode->sxTri.pnode2, byteCodeGenerator, prefix, postfix);

Visit(pnode->sxTri.pnode3, byteCodeGenerator, prefix, postfix);

break;

case knopList:

VisitList(pnode, byteCodeGenerator, prefix, postfix);

break;

// PTNODE(knopVarDecl , "varDcl" ,None ,Var ,fnopNone)

case knopVarDecl:

case knopConstDecl:

case knopLetDecl:

if (pnode->sxVar.pnodeInit != nullptr)

Visit(pnode->sxVar.pnodeInit, byteCodeGenerator, prefix, postfix);

break;

// PTNODE(knopFncDecl , "fncDcl" ,None ,Fnc ,fnopLeaf)

case knopFncDecl:

{

// Inner function declarations are visited before anything else in the scope.

// (See VisitFunctionsInScope.)

break;

}

case knopClassDecl:

{

// Visit the extends expression first, since it's bound outside the scope containing the class name.

Visit(pnode->sxClass.pnodeExtends, byteCodeGenerator, prefix, postfix);

Visit(pnode->sxClass.pnodeDeclName, byteCodeGenerator, prefix, postfix);

// Now visit the class name and methods.

BeginVisitBlock(pnode->sxClass.pnodeBlock, byteCodeGenerator);

Visit(pnode->sxClass.pnodeName, byteCodeGenerator, prefix, postfix);

Visit(pnode->sxClass.pnodeStaticMembers, byteCodeGenerator, prefix, postfix);

Visit(pnode->sxClass.pnodeConstructor, byteCodeGenerator, prefix, postfix);

Visit(pnode->sxClass.pnodeMembers, byteCodeGenerator, prefix, postfix);

EndVisitBlock(pnode->sxClass.pnodeBlock, byteCodeGenerator);

break;

}

case knopStrTemplate:

{

// Visit the string node lists only if we do not have a tagged template.

// We never need to visit the raw strings as they are not used in non-tagged templates and

// tagged templates will register them as part of the callsite constant object.

if (!pnode->sxStrTemplate.isTaggedTemplate)

{

Visit(pnode->sxStrTemplate.pnodeStringLiterals, byteCodeGenerator, prefix, postfix);

}

Visit(pnode->sxStrTemplate.pnodeSubstitutionExpressions, byteCodeGenerator, prefix, postfix);

break;

}

// PTNODE(knopProg , "program" ,None ,Fnc ,fnopNone)

case knopProg:

{

// We expect that the global statements have been generated (meaning that the pnodeFncs

// field is a real pointer, not an enumeration).

Assert(pnode->sxFnc.pnodeBody);

uint i = 0;

VisitNestedScopes(pnode->sxFnc.pnodeScopes, pnode, byteCodeGenerator, prefix, postfix, &i);

// Visiting global code: track the last value statement.

BeginVisitBlock(pnode->sxFnc.pnodeScopes, byteCodeGenerator);

pnode->sxProg.pnodeLastValStmt = VisitBlock(pnode->sxFnc.pnodeBody, byteCodeGenerator, prefix, postfix);

EndVisitBlock(pnode->sxFnc.pnodeScopes, byteCodeGenerator);

break;

}

case knopFor:

BeginVisitBlock(pnode->sxFor.pnodeBlock, byteCodeGenerator);

Visit(pnode->sxFor.pnodeInit, byteCodeGenerator, prefix, postfix);

byteCodeGenerator->EnterLoop();

Visit(pnode->sxFor.pnodeCond, byteCodeGenerator, prefix, postfix);

Visit(pnode->sxFor.pnodeIncr, byteCodeGenerator, prefix, postfix);

Visit(pnode->sxFor.pnodeBody, byteCodeGenerator, prefix, postfix, pnode);

byteCodeGenerator->ExitLoop();

EndVisitBlock(pnode->sxFor.pnodeBlock, byteCodeGenerator);

break;

// PTNODE(knopIf , "if" ,None ,If ,fnopNone)

case knopIf:

Visit(pnode->sxIf.pnodeCond, byteCodeGenerator, prefix, postfix);

Visit(pnode->sxIf.pnodeTrue, byteCodeGenerator, prefix, postfix, pnode);

if (pnode->sxIf.pnodeFalse != nullptr)

{

Visit(pnode->sxIf.pnodeFalse, byteCodeGenerator, prefix, postfix, pnode);

}

break;

// PTNODE(knopWhile , "while" ,None ,While,fnopBreak|fnopContinue)

// PTNODE(knopDoWhile , "do-while" ,None ,While,fnopBreak|fnopContinue)

case knopDoWhile:

case knopWhile:

byteCodeGenerator->EnterLoop();

Visit(pnode->sxWhile.pnodeCond, byteCodeGenerator, prefix, postfix);

Visit(pnode->sxWhile.pnodeBody, byteCodeGenerator, prefix, postfix, pnode);

byteCodeGenerator->ExitLoop();

break;

// PTNODE(knopForIn , "for in" ,None ,ForIn,fnopBreak|fnopContinue|fnopCleanup)

case knopForIn:

case knopForOf:

BeginVisitBlock(pnode->sxForInOrForOf.pnodeBlock, byteCodeGenerator);

Visit(pnode->sxForInOrForOf.pnodeLval, byteCodeGenerator, prefix, postfix);

Visit(pnode->sxForInOrForOf.pnodeObj, byteCodeGenerator, prefix, postfix);

byteCodeGenerator->EnterLoop();

Visit(pnode->sxForInOrForOf.pnodeBody, byteCodeGenerator, prefix, postfix, pnode);

byteCodeGenerator->ExitLoop();

EndVisitBlock(pnode->sxForInOrForOf.pnodeBlock, byteCodeGenerator);

break;

// PTNODE(knopReturn , "return" ,None ,Uni ,fnopNone)

case knopReturn:

if (pnode->sxReturn.pnodeExpr != nullptr)

Visit(pnode->sxReturn.pnodeExpr, byteCodeGenerator, prefix, postfix);

break;

// PTNODE(knopBlock , "{}" ,None ,Block,fnopNone)

case knopBlock:

{

if (pnode->sxBlock.pnodeStmt != nullptr)

{

BeginVisitBlock(pnode, byteCodeGenerator);

pnode->sxBlock.pnodeLastValStmt = VisitBlock(pnode->sxBlock.pnodeStmt, byteCodeGenerator, prefix, postfix, pnode);

EndVisitBlock(pnode, byteCodeGenerator);

}

else

{

pnode->sxBlock.pnodeLastValStmt = nullptr;

}

break;

}

// PTNODE(knopWith , "with" ,None ,With ,fnopCleanup)

case knopWith:

Visit(pnode->sxWith.pnodeObj, byteCodeGenerator, prefix, postfix);

VisitWithStmt(pnode, pnode->sxWith.pnodeObj->location, byteCodeGenerator, prefix, postfix, pnode);

break;

// PTNODE(knopBreak , "break" ,None ,Jump ,fnopNone)

case knopBreak:

// TODO: some representation of target

break;

// PTNODE(knopContinue , "continue" ,None ,Jump ,fnopNone)

case knopContinue:

// TODO: some representation of target

break;

// PTNODE(knopLabel , "label" ,None ,Label,fnopNone)

case knopLabel:

// TODO: print labeled statement

break;

// PTNODE(knopSwitch , "switch" ,None ,Switch,fnopBreak)

case knopSwitch:

Visit(pnode->sxSwitch.pnodeVal, byteCodeGenerator, prefix, postfix);

BeginVisitBlock(pnode->sxSwitch.pnodeBlock, byteCodeGenerator);

for (ParseNode \*pnodeT = pnode->sxSwitch.pnodeCases; nullptr != pnodeT; pnodeT = pnodeT->sxCase.pnodeNext)

{

Visit(pnodeT, byteCodeGenerator, prefix, postfix, pnode);

}

Visit(pnode->sxSwitch.pnodeBlock, byteCodeGenerator, prefix, postfix);

EndVisitBlock(pnode->sxSwitch.pnodeBlock, byteCodeGenerator);

break;

// PTNODE(knopCase , "case" ,None ,Case ,fnopNone)

case knopCase:

Visit(pnode->sxCase.pnodeExpr, byteCodeGenerator, prefix, postfix);

Visit(pnode->sxCase.pnodeBody, byteCodeGenerator, prefix, postfix, pnode);

break;

case knopTypeof:

Visit(pnode->sxUni.pnode1, byteCodeGenerator, prefix, postfix);

break;

// PTNODE(knopTryCatchFinally,"try-catch-finally",None,TryCatchFinally,fnopCleanup)

case knopTryFinally:

Visit(pnode->sxTryFinally.pnodeTry, byteCodeGenerator, prefix, postfix, pnode);

Visit(pnode->sxTryFinally.pnodeFinally, byteCodeGenerator, prefix, postfix, pnode);

break;

// PTNODE(knopTryCatch , "try-catch" ,None ,TryCatch ,fnopCleanup)

case knopTryCatch:

Visit(pnode->sxTryCatch.pnodeTry, byteCodeGenerator, prefix, postfix, pnode);

Visit(pnode->sxTryCatch.pnodeCatch, byteCodeGenerator, prefix, postfix, pnode);

break;

// PTNODE(knopTry , "try" ,None ,Try ,fnopCleanup)

case knopTry:

Visit(pnode->sxTry.pnodeBody, byteCodeGenerator, prefix, postfix, pnode);

break;

case knopCatch:

BeginVisitCatch(pnode, byteCodeGenerator);

Visit(pnode->sxCatch.pnodeParam, byteCodeGenerator, prefix, postfix);

Visit(pnode->sxCatch.pnodeBody, byteCodeGenerator, prefix, postfix, pnode);

EndVisitCatch(pnode, byteCodeGenerator);

break;

case knopFinally:

Visit(pnode->sxFinally.pnodeBody, byteCodeGenerator, prefix, postfix, pnode);

break;

// PTNODE(knopThrow , "throw" ,None ,Uni ,fnopNone)

case knopThrow:

Visit(pnode->sxUni.pnode1, byteCodeGenerator, prefix, postfix);

break;

case knopArray:

{

bool arrayLitOpt = EmitAsConstantArray(pnode, byteCodeGenerator);

if (!arrayLitOpt)

{

Visit(pnode->sxUni.pnode1, byteCodeGenerator, prefix, postfix);

}

break;

}

case knopComma:

{

ParseNode \*pnode1 = pnode->sxBin.pnode1;

if (pnode1->nop == knopComma)

{

// Spot-fix to avoid recursion on very large comma expressions.

ArenaAllocator \*alloc = byteCodeGenerator->GetAllocator();

SList<ParseNode\*> rhsStack(alloc);

do

{

rhsStack.Push(pnode1->sxBin.pnode2);

pnode1 = pnode1->sxBin.pnode1;

}

while (pnode1->nop == knopComma);

Visit(pnode1, byteCodeGenerator, prefix, postfix);

while (!rhsStack.Empty())

{

ParseNode \*pnodeRhs = rhsStack.Pop();

Visit(pnodeRhs, byteCodeGenerator, prefix, postfix);

}

}

else

{

Visit(pnode1, byteCodeGenerator, prefix, postfix);

}

Visit(pnode->sxBin.pnode2, byteCodeGenerator, prefix, postfix);

}

break;

}

if (pnodeParent)

{

PropagateFlags(pnode, pnodeParent);

}

postfix(pnode, byteCodeGenerator);

}

bool IsJump(ParseNode \*pnode)

{

switch (pnode->nop)

{

case knopBreak:

case knopContinue:

case knopThrow:

case knopReturn:

return true;

case knopBlock:

case knopDoWhile:

case knopWhile:

case knopWith:

case knopIf:

case knopForIn:

case knopForOf:

case knopFor:

case knopSwitch:

case knopCase:

case knopTryFinally:

case knopTryCatch:

case knopTry:

case knopCatch:

case knopFinally:

return (pnode->sxStmt.grfnop & fnopJump) != 0;

default:

return false;

}

}

void PropagateFlags(ParseNode \*pnodeChild, ParseNode \*pnodeParent)

{

if (IsJump(pnodeChild))

{

pnodeParent->sxStmt.grfnop |= fnopJump;

}

}

void Bind(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator);

void BindReference(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator);

void AssignRegisters(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator);

// TODO[ianhall]: This should be in a shared AST Utility header or source file

bool IsExpressionStatement(ParseNode\* stmt, const Js::ScriptContext \*const scriptContext)

{

if (stmt->nop == knopFncDecl)

{

// 'knopFncDecl' is used for both function declarations and function expressions. In a program, a function expression

// produces the function object that is created for the function expression as its value for the program. A function

// declaration does not produce a value for the program.

return !stmt->sxFnc.IsDeclaration();

}

if ((stmt->nop >= 0) && (stmt->nop<knopLim))

{

return (ParseNode::Grfnop(stmt->nop) & fnopNotExprStmt) == 0;

}

return false;

}

bool MustProduceValue(ParseNode \*pnode, const Js::ScriptContext \*const scriptContext)

{

// Determine whether the current statement is guaranteed to produce a value.

if (IsExpressionStatement(pnode, scriptContext))

{

// These are trivially true.

return true;

}

for (;;)

{

switch (pnode->nop)

{

case knopFor:

// Check the common "for (;;)" case.

if (pnode->sxFor.pnodeCond != nullptr ||

pnode->sxFor.pnodeBody == nullptr)

{

return false;

}

// Loop body is always executed. Look at the loop body next.

pnode = pnode->sxFor.pnodeBody;

break;

case knopIf:

// True only if both "if" and "else" exist, and both produce values.

if (pnode->sxIf.pnodeTrue == nullptr ||

pnode->sxIf.pnodeFalse == nullptr)

{

return false;

}

if (!MustProduceValue(pnode->sxIf.pnodeFalse, scriptContext))

{

return false;

}

pnode = pnode->sxIf.pnodeTrue;

break;

case knopWhile:

// Check the common "while (1)" case.

if (pnode->sxWhile.pnodeBody == nullptr ||

(pnode->sxWhile.pnodeCond &&

(pnode->sxWhile.pnodeCond->nop != knopInt ||

pnode->sxWhile.pnodeCond->sxInt.lw == 0)))

{

return false;

}

// Loop body is always executed. Look at the loop body next.

pnode = pnode->sxWhile.pnodeBody;

break;

case knopDoWhile:

if (pnode->sxWhile.pnodeBody == nullptr)

{

return false;

}

// Loop body is always executed. Look at the loop body next.

pnode = pnode->sxWhile.pnodeBody;

break;

case knopBlock:

return pnode->sxBlock.pnodeLastValStmt != nullptr;

case knopWith:

if (pnode->sxWith.pnodeBody == nullptr)

{

return false;

}

pnode = pnode->sxWith.pnodeBody;

break;

case knopSwitch:

{

// This is potentially the most inefficient case. We could consider adding a flag to the PnSwitch

// struct and computing it when we visit the switch, but:

// a. switch statements at global scope shouldn't be that common;

// b. switch statements with many arms shouldn't be that common;

// c. switches without default cases can be trivially skipped.

if (pnode->sxSwitch.pnodeDefault == nullptr)

{

// Can't guarantee that any code is executed.

return false;

}

ParseNode \*pnodeCase;

for (pnodeCase = pnode->sxSwitch.pnodeCases; pnodeCase; pnodeCase = pnodeCase->sxCase.pnodeNext)

{

if (pnodeCase->sxCase.pnodeBody == nullptr)

{

if (pnodeCase->sxCase.pnodeNext == nullptr)

{

// Last case has no code to execute.

return false;

}

// Fall through to the next case.

}

else

{

if (!MustProduceValue(pnodeCase->sxCase.pnodeBody, scriptContext))

{

return false;

}

}

}

return true;

}

case knopTryCatch:

// True only if both try and catch produce a value.

if (pnode->sxTryCatch.pnodeTry->sxTry.pnodeBody == nullptr ||

pnode->sxTryCatch.pnodeCatch->sxCatch.pnodeBody == nullptr)

{

return false;

}

if (!MustProduceValue(pnode->sxTryCatch.pnodeCatch->sxCatch.pnodeBody, scriptContext))

{

return false;

}

pnode = pnode->sxTryCatch.pnodeTry->sxTry.pnodeBody;

break;

case knopTryFinally:

if (pnode->sxTryFinally.pnodeFinally->sxFinally.pnodeBody == nullptr)

{

// No finally body: look at the try body.

if (pnode->sxTryFinally.pnodeTry->sxTry.pnodeBody == nullptr)

{

return false;

}

pnode = pnode->sxTryFinally.pnodeTry->sxTry.pnodeBody;

break;

}

// Skip the try body, since the finally body will always follow it.

pnode = pnode->sxTryFinally.pnodeFinally->sxFinally.pnodeBody;

break;

default:

return false;

}

}

}

ByteCodeGenerator::ByteCodeGenerator(Js::ScriptContext\* scriptContext, Js::ScopeInfo\* parentScopeInfo) :

alloc(nullptr),

scriptContext(scriptContext),

flags(0),

funcInfoStack(nullptr),

pRootFunc(nullptr),

pCurrentFunction(nullptr),

globalScope(nullptr),

currentScope(nullptr),

parentScopeInfo(parentScopeInfo),

dynamicScopeCount(0),

isBinding(false),

propertyRecords(nullptr),

inDestructuredPattern(false)

{

m\_writer.Create();

}

/\* static \*/

bool ByteCodeGenerator::IsFalse(ParseNode\* node)

{

return (node->nop == knopInt && node->sxInt.lw == 0) || node->nop == knopFalse;

}

bool ByteCodeGenerator::UseParserBindings() const

{

return IsInNonDebugMode() && !PHASE\_OFF1(Js::ParserBindPhase);

}

bool ByteCodeGenerator::IsES6DestructuringEnabled() const

{

return scriptContext->GetConfig()->IsES6DestructuringEnabled();

}

bool ByteCodeGenerator::IsES6ForLoopSemanticsEnabled() const

{

return scriptContext->GetConfig()->IsES6ForLoopSemanticsEnabled();

}

// ByteCodeGenerator debug mode means we are generating debug mode user-code. Library code is always in non-debug mode.

bool ByteCodeGenerator::IsInDebugMode() const

{

return scriptContext->IsInDebugMode() && !m\_utf8SourceInfo->GetIsLibraryCode();

}

// ByteCodeGenerator non-debug mode means we are not debugging, or we are generating library code which is always in non-debug mode.

bool ByteCodeGenerator::IsInNonDebugMode() const

{

return scriptContext->IsInNonDebugMode() || m\_utf8SourceInfo->GetIsLibraryCode();

}

bool ByteCodeGenerator::ShouldTrackDebuggerMetadata() const

{

return (IsInDebugMode())

#if DBG\_DUMP

|| (Js::Configuration::Global.flags.Debug)

#endif

;

}

void ByteCodeGenerator::SetRootFuncInfo(FuncInfo\* func)

{

Assert(pRootFunc == nullptr || pRootFunc == func->byteCodeFunction || !IsInNonDebugMode());

if (this->flags & (fscrImplicitThis | fscrImplicitParents))

{

// Mark a top-level event handler, since it will need to construct the "this" pointer's

// namespace hierarchy to access globals.

Assert(!func->IsGlobalFunction());

func->SetIsEventHandler(true);

}

if (pRootFunc)

{

return;

}

this->pRootFunc = func->byteCodeFunction->GetParseableFunctionInfo();

}

Js::RegSlot ByteCodeGenerator::NextVarRegister()

{

return funcInfoStack->Top()->NextVarRegister();

}

Js::RegSlot ByteCodeGenerator::NextConstRegister()

{

return funcInfoStack->Top()->NextConstRegister();

}

FuncInfo \* ByteCodeGenerator::TopFuncInfo() const

{

return funcInfoStack->Empty() ? nullptr : funcInfoStack->Top();

}

void ByteCodeGenerator::EnterLoop()

{

if (this->TopFuncInfo())

{

this->TopFuncInfo()->hasLoop = true;

}

loopDepth++;

}

void ByteCodeGenerator::SetHasTry(bool has)

{

TopFuncInfo()->GetParsedFunctionBody()->SetHasTry(has);

}

void ByteCodeGenerator::SetHasFinally(bool has)

{

TopFuncInfo()->GetParsedFunctionBody()->SetHasFinally(has);

}

// TODO: per-function register assignment for env and global symbols

void ByteCodeGenerator::AssignRegister(Symbol \*sym)

{

AssertMsg(sym->GetDecl() == nullptr || sym->GetDecl()->nop != knopConstDecl || sym->GetDecl()->nop != knopLetDecl,

"const and let should get only temporary register, assigned during emit stage");

if (sym->GetLocation() == Js::Constants::NoRegister)

{

sym->SetLocation(NextVarRegister());

}

}

void ByteCodeGenerator::AddTargetStmt(ParseNode \*pnodeStmt)

{

FuncInfo \*top = funcInfoStack->Top();

top->AddTargetStmt(pnodeStmt);

}

Js::RegSlot ByteCodeGenerator::AssignNullConstRegister()

{

FuncInfo \*top = funcInfoStack->Top();

return top->AssignNullConstRegister();

}

Js::RegSlot ByteCodeGenerator::AssignUndefinedConstRegister()

{

FuncInfo \*top = funcInfoStack->Top();

return top->AssignUndefinedConstRegister();

}

Js::RegSlot ByteCodeGenerator::AssignTrueConstRegister()

{

FuncInfo \*top = funcInfoStack->Top();

return top->AssignTrueConstRegister();

}

Js::RegSlot ByteCodeGenerator::AssignFalseConstRegister()

{

FuncInfo \*top = funcInfoStack->Top();

return top->AssignFalseConstRegister();

}

Js::RegSlot ByteCodeGenerator::AssignThisRegister()

{

FuncInfo \*top = funcInfoStack->Top();

return top->AssignThisRegister();

}

Js::RegSlot ByteCodeGenerator::AssignNewTargetRegister()

{

FuncInfo \*top = funcInfoStack->Top();

return top->AssignNewTargetRegister();

}

void ByteCodeGenerator::SetNeedEnvRegister()

{

FuncInfo \*top = funcInfoStack->Top();

top->SetNeedEnvRegister();

}

void ByteCodeGenerator::AssignFrameObjRegister()

{

FuncInfo\* top = funcInfoStack->Top();

if (top->frameObjRegister == Js::Constants::NoRegister)

{

top->frameObjRegister = top->NextVarRegister();

}

}

void ByteCodeGenerator::AssignFrameDisplayRegister()

{

FuncInfo\* top = funcInfoStack->Top();

if (top->frameDisplayRegister == Js::Constants::NoRegister)

{

top->frameDisplayRegister = top->NextVarRegister();

}

}

void ByteCodeGenerator::AssignFrameSlotsRegister()

{

FuncInfo\* top = funcInfoStack->Top();

if (top->frameSlotsRegister == Js::Constants::NoRegister)

{

top->frameSlotsRegister = NextVarRegister();

}

}

void ByteCodeGenerator::SetNumberOfInArgs(Js::ArgSlot argCount)

{

FuncInfo \*top = funcInfoStack->Top();

top->inArgsCount = argCount;

}

Js::RegSlot ByteCodeGenerator::EnregisterConstant(unsigned int constant)

{

Js::RegSlot loc = Js::Constants::NoRegister;

FuncInfo \*top = funcInfoStack->Top();

if (!top->constantToRegister.TryGetValue(constant, &loc))

{

loc = NextConstRegister();

top->constantToRegister.Add(constant, loc);

}

return loc;

}

Js::RegSlot ByteCodeGenerator::EnregisterStringConstant(IdentPtr pid)

{

Js::RegSlot loc = Js::Constants::NoRegister;

FuncInfo \*top = funcInfoStack->Top();

if (!top->stringToRegister.TryGetValue(pid, &loc))

{

loc = NextConstRegister();

top->stringToRegister.Add(pid, loc);

}

return loc;

}

Js::RegSlot ByteCodeGenerator::EnregisterDoubleConstant(double d)

{

Js::RegSlot loc = Js::Constants::NoRegister;

FuncInfo \*top = funcInfoStack->Top();

if (!top->TryGetDoubleLoc(d, &loc))

{

loc = NextConstRegister();

top->AddDoubleConstant(d, loc);

}

return loc;

}

Js::RegSlot ByteCodeGenerator::EnregisterStringTemplateCallsiteConstant(ParseNode\* pnode)

{

Assert(pnode->nop == knopStrTemplate);

Assert(pnode->sxStrTemplate.isTaggedTemplate);

Js::RegSlot loc = Js::Constants::NoRegister;

FuncInfo\* top = funcInfoStack->Top();

if (!top->stringTemplateCallsiteRegisterMap.TryGetValue(pnode, &loc))

{

loc = NextConstRegister();

top->stringTemplateCallsiteRegisterMap.Add(pnode, loc);

}

return loc;

}

//

// Restore all outer func scope info when reparsing a deferred func.

//

void ByteCodeGenerator::RestoreScopeInfo(Js::FunctionBody\* functionBody)

{

if (functionBody && functionBody->GetScopeInfo())

{

PROBE\_STACK(scriptContext, Js::Constants::MinStackByteCodeVisitor);

Js::ScopeInfo\* scopeInfo = functionBody->GetScopeInfo();

RestoreScopeInfo(scopeInfo->GetParent()); // Recursively restore outer func scope info

Js::ScopeInfo\* paramScopeInfo = scopeInfo->GetParamScopeInfo();

Scope\* paramScope = nullptr;

if (paramScopeInfo != nullptr)

{

paramScope = paramScopeInfo->GetScope();

Assert(paramScope || !UseParserBindings());

if (paramScope == nullptr || !UseParserBindings())

{

paramScope = Anew(alloc, Scope, alloc, ScopeType\_Parameter, true);

}

// We need the funcInfo before continuing the restoration of the param scope, so wait for the funcInfo to be created.

}

Scope\* bodyScope = scopeInfo->GetScope();

Assert(bodyScope || !UseParserBindings());

if (bodyScope == nullptr || !UseParserBindings())

{

if (scopeInfo->IsGlobalEval())

{

bodyScope = Anew(alloc, Scope, alloc, ScopeType\_GlobalEvalBlock, true);

}

else

{

bodyScope = Anew(alloc, Scope, alloc, ScopeType\_FunctionBody, true);

}

}

FuncInfo\* func = Anew(alloc, FuncInfo, functionBody->GetDisplayName(), alloc, paramScope, bodyScope, nullptr, functionBody);

if (paramScope != nullptr)

{

paramScope->SetFunc(func);

paramScopeInfo->GetScopeInfo(nullptr, this, func, paramScope);

}

if (bodyScope->GetScopeType() == ScopeType\_GlobalEvalBlock)

{

func->bodyScope = this->currentScope;

}

PushFuncInfo(L"RestoreScopeInfo", func);

if (!functionBody->DoStackNestedFunc())

{

func->hasEscapedUseNestedFunc = true;

}

Js::ScopeInfo\* funcExprScopeInfo = scopeInfo->GetFuncExprScopeInfo();

if (funcExprScopeInfo)

{

Scope\* funcExprScope = funcExprScopeInfo->GetScope();

Assert(funcExprScope || !UseParserBindings());

if (funcExprScope == nullptr || !UseParserBindings())

{

funcExprScope = Anew(alloc, Scope, alloc, ScopeType\_FuncExpr, true);

}

funcExprScope->SetFunc(func);

func->SetFuncExprScope(funcExprScope);

funcExprScopeInfo->GetScopeInfo(nullptr, this, func, funcExprScope);

}

scopeInfo->GetScopeInfo(nullptr, this, func, bodyScope);

}

else

{

Assert(this->TopFuncInfo() == nullptr);

// funcBody is glo

currentScope = Anew(alloc, Scope, alloc, ScopeType\_Global, !UseParserBindings());

globalScope = currentScope;

FuncInfo \*func = Anew(alloc, FuncInfo, Js::Constants::GlobalFunction,

alloc, nullptr, currentScope, nullptr, functionBody);

PushFuncInfo(L"RestoreScopeInfo", func);

}

}

FuncInfo \* ByteCodeGenerator::StartBindGlobalStatements(ParseNode \*pnode)

{

if (parentScopeInfo)

{

Assert(CONFIG\_FLAG(DeferNested));

trackEnvDepth = true;

RestoreScopeInfo(parentScopeInfo->GetParent());

trackEnvDepth = false;

// "currentScope" is the parentFunc scope. This ensures the deferred func declaration

// symbol will bind to the func declaration symbol already available in parentFunc scope.

}

else

{

currentScope = pnode->sxProg.scope;

Assert(currentScope);

if (!currentScope || !UseParserBindings())

{

currentScope = Anew(alloc, Scope, alloc, ScopeType\_Global, true);

pnode->sxProg.scope = currentScope;

}

globalScope = currentScope;

}

Js::FunctionBody \* byteCodeFunction;

if (!IsInNonDebugMode() && this->pCurrentFunction != nullptr && this->pCurrentFunction->GetIsGlobalFunc() && !this->pCurrentFunction->IsFakeGlobalFunc(flags))

{

// we will re-use the global FunctionBody which was created before deferred parse.

byteCodeFunction = this->pCurrentFunction;

byteCodeFunction->RemoveDeferParseAttribute();

byteCodeFunction->ResetByteCodeGenVisitState();

if (byteCodeFunction->GetBoundPropertyRecords() == nullptr)

{

Assert(!IsInNonDebugMode());

// This happens when we try to re-use the function body which was created due to serialized bytecode.

byteCodeFunction->SetBoundPropertyRecords(EnsurePropertyRecordList());

}

}

else if ((this->flags & fscrDeferredFnc))

{

byteCodeFunction = this->EnsureFakeGlobalFuncForUndefer(pnode);

}

else

{

byteCodeFunction = this->MakeGlobalFunctionBody(pnode);

// Mark this global function to required for register script event

byteCodeFunction->SetIsTopLevel(true);

if (pnode->sxFnc.GetStrictMode() != 0)

{

byteCodeFunction->SetIsStrictMode();

}

}

if (byteCodeFunction->IsReparsed())

{

byteCodeFunction->RestoreState(pnode);

}

else

{

byteCodeFunction->SaveState(pnode);

}

FuncInfo \*funcInfo = Anew(alloc, FuncInfo, Js::Constants::GlobalFunction,

alloc, nullptr, globalScope, pnode, byteCodeFunction);

long currentAstSize = pnode->sxFnc.astSize;

if (currentAstSize > this->maxAstSize)

{

this->maxAstSize = currentAstSize;

}

PushFuncInfo(L"StartBindGlobalStatements", funcInfo);

return funcInfo;

}

void ByteCodeGenerator::AssignPropertyId(IdentPtr pid)

{

if (pid->GetPropertyId() == Js::Constants::NoProperty)

{

Js::PropertyId id = TopFuncInfo()->byteCodeFunction->GetOrAddPropertyIdTracked(SymbolName(pid->Psz(), pid->Cch()));

pid->SetPropertyId(id);

}

}

void ByteCodeGenerator::AssignPropertyId(Symbol \*sym, Js::ParseableFunctionInfo\* functionInfo)

{

sym->SetPosition(functionInfo->GetOrAddPropertyIdTracked(sym->GetName()));

}

template <class PrefixFn, class PostfixFn>

ParseNode\* VisitBlock(ParseNode \*pnode, ByteCodeGenerator\* byteCodeGenerator, PrefixFn prefix, PostfixFn postfix, ParseNode \*pnodeParent = nullptr)

{

ParseNode \*pnodeLastVal = nullptr;

if (pnode != nullptr)

{

bool fTrackVal = byteCodeGenerator->IsBinding() &&

(byteCodeGenerator->GetFlags() & fscrReturnExpression) &&

byteCodeGenerator->TopFuncInfo()->IsGlobalFunction();

while (pnode->nop == knopList)

{

Visit(pnode->sxBin.pnode1, byteCodeGenerator, prefix, postfix, pnodeParent);

if (fTrackVal)

{

// If we're tracking values, find the last statement (if any) in the block that is

// guaranteed to produce a value.

if (MustProduceValue(pnode->sxBin.pnode1, byteCodeGenerator->GetScriptContext()))

{

pnodeLastVal = pnode->sxBin.pnode1;

}

if (IsJump(pnode->sxBin.pnode1))

{

// This is a jump out of the current block. The remaining instructions (if any)

// will not be executed, so stop tracking them.

fTrackVal = false;

}

}

pnode = pnode->sxBin.pnode2;

}

Visit(pnode, byteCodeGenerator, prefix, postfix, pnodeParent);

if (fTrackVal)

{

if (MustProduceValue(pnode, byteCodeGenerator->GetScriptContext()))

{

pnodeLastVal = pnode;

}

}

}

return pnodeLastVal;

}

FuncInfo \* ByteCodeGenerator::StartBindFunction(const wchar\_t \*name, uint nameLength, uint shortNameOffset, bool\* pfuncExprWithName, ParseNode \*pnode)

{

bool funcExprWithName;

union

{

Js::ParseableFunctionInfo\* parseableFunctionInfo;

Js::FunctionBody\* parsedFunctionBody;

};

bool isDeferParsed = false;

if (this->pCurrentFunction &&

this->pCurrentFunction->IsFunctionParsed())

{

Assert(this->pCurrentFunction->StartInDocument() == pnode->ichMin);

Assert(this->pCurrentFunction->LengthInChars() == pnode->LengthInCodepoints());

// This is the root function for the current AST subtree, and it already has a FunctionBody

// (created by a deferred parse) which we're now filling in.

parsedFunctionBody = this->pCurrentFunction;

parsedFunctionBody->RemoveDeferParseAttribute();

if (parsedFunctionBody->GetBoundPropertyRecords() == nullptr)

{

// This happens when we try to re-use the function body which was created due to serialized bytecode.

parsedFunctionBody->SetBoundPropertyRecords(EnsurePropertyRecordList());

}

Assert(!parsedFunctionBody->IsDeferredParseFunction() || parsedFunctionBody->IsReparsed());

pnode->sxFnc.SetDeclaration(parsedFunctionBody->GetIsDeclaration());

funcExprWithName =

!(parsedFunctionBody->GetIsDeclaration() || pnode->sxFnc.IsMethod()) &&

pnode->sxFnc.pnodeName != nullptr &&

pnode->sxFnc.pnodeName->nop == knopVarDecl;

\*pfuncExprWithName = funcExprWithName;

Assert(parsedFunctionBody->GetLocalFunctionId() == pnode->sxFnc.functionId || !IsInNonDebugMode());

// Some state may be tracked on the function body during the visit pass. Since the previous visit pass may have failed,

// we need to reset the state on the function body.

parsedFunctionBody->ResetByteCodeGenVisitState();

if (parsedFunctionBody->GetScopeInfo())

{

// Propagate flags from the (real) parent function.

Js::ParseableFunctionInfo \*parent = parsedFunctionBody->GetScopeInfo()->GetParent();

if (parent)

{

Assert(parent->GetFunctionBody());

if (parent->GetFunctionBody()->GetHasOrParentHasArguments())

{

parsedFunctionBody->SetHasOrParentHasArguments(true);

}

}

}

}

else

{

funcExprWithName = \*pfuncExprWithName;

Js::LocalFunctionId functionId = pnode->sxFnc.functionId;

isDeferParsed = (pnode->sxFnc.pnodeBody == nullptr);

// Create a function body if:

// 1. The parse node is not defer parsed

// 2. Or creating function proxies is disallowed

bool createFunctionBody = !isDeferParsed;

if (!CONFIG\_FLAG(CreateFunctionProxy)) createFunctionBody = true;

Js::FunctionInfo::Attributes attributes = Js::FunctionInfo::Attributes::None;

if (pnode->sxFnc.IsAsync())

{

attributes = (Js::FunctionInfo::Attributes)(attributes | Js::FunctionInfo::Attributes::ErrorOnNew | Js::FunctionInfo::Attributes::Async);

}

if (pnode->sxFnc.IsLambda())

{

attributes = (Js::FunctionInfo::Attributes)(attributes | Js::FunctionInfo::Attributes::ErrorOnNew | Js::FunctionInfo::Attributes::Lambda);

}

if (pnode->sxFnc.HasSuperReference())

{

attributes = (Js::FunctionInfo::Attributes)(attributes | Js::FunctionInfo::Attributes::SuperReference);

}

if (pnode->sxFnc.IsClassMember())

{

if (pnode->sxFnc.IsClassConstructor())

{

attributes = (Js::FunctionInfo::Attributes)(attributes | Js::FunctionInfo::Attributes::ClassConstructor);

if (pnode->sxFnc.IsGeneratedDefault())

{

attributes = (Js::FunctionInfo::Attributes)(attributes | Js::FunctionInfo::Attributes::DefaultConstructor);

}

}

else

{

attributes = (Js::FunctionInfo::Attributes)(attributes | Js::FunctionInfo::Attributes::ErrorOnNew | Js::FunctionInfo::Attributes::ClassMethod);

}

}

if (pnode->sxFnc.IsGenerator())

{

attributes = (Js::FunctionInfo::Attributes)(attributes | Js::FunctionInfo::Attributes::Generator);

}

if (pnode->sxFnc.IsAccessor())

{

attributes = (Js::FunctionInfo::Attributes)(attributes | Js::FunctionInfo::Attributes::ErrorOnNew);

}

if (createFunctionBody)

{

ENTER\_PINNED\_SCOPE(Js::PropertyRecordList, propertyRecordList);

propertyRecordList = EnsurePropertyRecordList();

parsedFunctionBody = Js::FunctionBody::NewFromRecycler(scriptContext, name, nameLength, shortNameOffset, pnode->sxFnc.nestedCount, m\_utf8SourceInfo,

m\_utf8SourceInfo->GetSrcInfo()->sourceContextInfo->sourceContextId, functionId, propertyRecordList

, attributes

#ifdef PERF\_COUNTERS

, false /\* is function from deferred deserialized proxy \*/

#endif

);

LEAVE\_PINNED\_SCOPE();

}

else

{

ENTER\_PINNED\_SCOPE(Js::PropertyRecordList, propertyRecordList);

propertyRecordList = nullptr;

if (funcExprWithName)

{

propertyRecordList = EnsurePropertyRecordList();

}

parseableFunctionInfo = Js::ParseableFunctionInfo::New(scriptContext, pnode->sxFnc.nestedCount, functionId, m\_utf8SourceInfo, name, nameLength, shortNameOffset, propertyRecordList, attributes);

LEAVE\_PINNED\_SCOPE();

}

// In either case register the function reference

scriptContext->RegisterDynamicFunctionReference(parseableFunctionInfo);

#if DBG

parseableFunctionInfo->deferredParseNextFunctionId = pnode->sxFnc.deferredParseNextFunctionId;

#endif

parseableFunctionInfo->SetIsDeclaration(pnode->sxFnc.IsDeclaration() != 0);

parseableFunctionInfo->SetIsAccessor(pnode->sxFnc.IsAccessor() != 0);

if (pnode->sxFnc.IsAccessor())

{

scriptContext->optimizationOverrides.SetSideEffects(Js::SideEffects\_Accessor);

}

}

Scope \*funcExprScope = nullptr;

if (funcExprWithName)

{

if (!UseParserBindings())

{

funcExprScope = Anew(alloc, Scope, alloc, ScopeType\_FuncExpr, true);

pnode->sxFnc.scope = funcExprScope;

}

else

{

funcExprScope = pnode->sxFnc.scope;

Assert(funcExprScope);

}

PushScope(funcExprScope);

Symbol \*sym = AddSymbolToScope(funcExprScope, name, nameLength, pnode->sxFnc.pnodeName, STFunction);

sym->SetFuncExpr(true);

sym->SetPosition(parsedFunctionBody->GetOrAddPropertyIdTracked(sym->GetName()));

pnode->sxFnc.SetFuncSymbol(sym);

}

Scope \*paramScope = pnode->sxFnc.pnodeScopes ? pnode->sxFnc.pnodeScopes->sxBlock.scope : nullptr;

Scope \*bodyScope = pnode->sxFnc.pnodeBodyScope ? pnode->sxFnc.pnodeBodyScope->sxBlock.scope : nullptr;

Assert(paramScope != nullptr || !pnode->sxFnc.pnodeScopes || !UseParserBindings());

if (paramScope == nullptr || !UseParserBindings())

{

paramScope = Anew(alloc, Scope, alloc, ScopeType\_Parameter, true);

if (pnode->sxFnc.pnodeScopes)

{

pnode->sxFnc.pnodeScopes->sxBlock.scope = paramScope;

}

}

if (bodyScope == nullptr || !UseParserBindings())

{

bodyScope = Anew(alloc, Scope, alloc, ScopeType\_FunctionBody, true);

if (pnode->sxFnc.pnodeBodyScope)

{

pnode->sxFnc.pnodeBodyScope->sxBlock.scope = bodyScope;

}

}

AssertMsg(pnode->nop == knopFncDecl, "Non-function declaration trying to create function body");

parseableFunctionInfo->SetIsGlobalFunc(false);

if (pnode->sxFnc.GetStrictMode() != 0)

{

parseableFunctionInfo->SetIsStrictMode();

}

FuncInfo \*funcInfo = Anew(alloc, FuncInfo, name, alloc, paramScope, bodyScope, pnode, parseableFunctionInfo);

if (pnode->sxFnc.GetArgumentsObjectEscapes())

{

// If the parser detected that the arguments object escapes, then the function scope escapes

// and cannot be cached.

this->FuncEscapes(bodyScope);

funcInfo->SetHasMaybeEscapedNestedFunc(DebugOnly(L"ArgumentsObjectEscapes"));

}

if (!isDeferParsed)

{

if (parsedFunctionBody->IsReparsed())

{

parsedFunctionBody->RestoreState(pnode);

}

else

{

parsedFunctionBody->SaveState(pnode);

}

}

funcInfo->SetChildCallsEval(!!pnode->sxFnc.ChildCallsEval());

if (pnode->sxFnc.CallsEval())

{

funcInfo->SetCallsEval(true);

bodyScope->SetIsDynamic(true);

bodyScope->SetIsObject();

bodyScope->SetCapturesAll(true);

bodyScope->SetMustInstantiate(true);

paramScope->SetIsObject();

paramScope->SetMustInstantiate(true);

paramScope->SetCapturesAll(true);

}

PushFuncInfo(L"StartBindFunction", funcInfo);

PushScope(paramScope);

PushScope(bodyScope);

if (funcExprScope)

{

funcExprScope->SetFunc(funcInfo);

funcInfo->funcExprScope = funcExprScope;

}

long currentAstSize = pnode->sxFnc.astSize;

if (currentAstSize > this->maxAstSize)

{

this->maxAstSize = currentAstSize;

}

return funcInfo;

}

void ByteCodeGenerator::EndBindFunction(bool funcExprWithName)

{

bool isGlobalScope = currentScope->GetScopeType() == ScopeType\_Global;

Assert(currentScope->GetScopeType() == ScopeType\_FunctionBody || isGlobalScope);

PopScope(); // function body

if (isGlobalScope)

{

Assert(currentScope == nullptr);

}

else

{

Assert(currentScope->GetScopeType() == ScopeType\_Parameter);

PopScope(); // parameter scope

}

if (funcExprWithName)

{

Assert(currentScope->GetScopeType() == ScopeType\_FuncExpr);

PopScope();

}

funcInfoStack->Pop();

}

void ByteCodeGenerator::StartBindCatch(ParseNode \*pnode)

{

Scope \*scope = pnode->sxCatch.scope;

Assert(scope);

if (scope == nullptr || !UseParserBindings())

{

scope = Anew(alloc, Scope, alloc, (pnode->sxCatch.pnodeParam->nop == knopParamPattern) ? ScopeType\_CatchParamPattern : ScopeType\_Catch, true);

pnode->sxCatch.scope = scope;

}

Assert(currentScope);

scope->SetFunc(currentScope->GetFunc());

PushScope(scope);

}

void ByteCodeGenerator::EndBindCatch()

{

PopScope();

}

void ByteCodeGenerator::PushScope(Scope \*innerScope)

{

Assert(innerScope != nullptr);

if (isBinding

&& currentScope != nullptr

&& currentScope->GetScopeType() == ScopeType\_FunctionBody

&& innerScope->GetMustInstantiate())

{

// If the current scope is a function body, we don't expect another function body

// without going through a function expression or parameter scope first. This may

// not be the case in the emit phase, where we can merge the two scopes. This also

// does not apply to incoming scopes marked as !mustInstantiate.

Assert(innerScope->GetScopeType() != ScopeType\_FunctionBody);

}

innerScope->SetEnclosingScope(currentScope);

currentScope = innerScope;

if (currentScope->GetIsDynamic())

{

this->dynamicScopeCount++;

}

if (this->trackEnvDepth && currentScope->GetMustInstantiate())

{

this->envDepth++;

if (this->envDepth == 0)

{

Js::Throw::OutOfMemory();

}

}

}

void ByteCodeGenerator::PopScope()

{

Assert(currentScope != nullptr);

if (this->trackEnvDepth && currentScope->GetMustInstantiate())

{

this->envDepth--;

Assert(this->envDepth != (uint16)-1);

}

if (currentScope->GetIsDynamic())

{

this->dynamicScopeCount--;

}

currentScope = currentScope->GetEnclosingScope();

}

void ByteCodeGenerator::PushBlock(ParseNode \*pnode)

{

pnode->sxBlock.SetEnclosingBlock(currentBlock);

currentBlock = pnode;

}

void ByteCodeGenerator::PopBlock()

{

currentBlock = currentBlock->sxBlock.GetEnclosingBlock();

}

void ByteCodeGenerator::PushFuncInfo(wchar\_t const \* location, FuncInfo\* funcInfo)

{

// We might have multiple global scope for deferparse.

// Assert(!funcInfo->IsGlobalFunction() || this->TopFuncInfo() == nullptr || this->TopFuncInfo()->IsGlobalFunction());

if (PHASE\_TRACE1(Js::ByteCodePhase))

{

Output::Print(L"%s: PushFuncInfo: %s", location, funcInfo->name);

if (this->TopFuncInfo())

{

Output::Print(L" Top: %s", this->TopFuncInfo()->name);

}

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

Output::Flush();

}

funcInfoStack->Push(funcInfo);

}

void ByteCodeGenerator::PopFuncInfo(wchar\_t const \* location)

{

FuncInfo \* funcInfo = funcInfoStack->Pop();

// Assert(!funcInfo->IsGlobalFunction() || this->TopFuncInfo() == nullptr || this->TopFuncInfo()->IsGlobalFunction());

if (PHASE\_TRACE1(Js::ByteCodePhase))

{

Output::Print(L"%s: PopFuncInfo: %s", location, funcInfo->name);

if (this->TopFuncInfo())

{

Output::Print(L" Top: %s", this->TopFuncInfo()->name);

}

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

Output::Flush();

}

}

Symbol \* ByteCodeGenerator::FindSymbol(Symbol \*\*symRef, IdentPtr pid, bool forReference)

{

const wchar\_t \*key = nullptr;

int keyLength;

Symbol \*sym = nullptr;

if (!UseParserBindings())

{

key = reinterpret\_cast<const wchar\_t\*>(pid->Psz());

keyLength = pid->Cch();

sym = currentScope->FindSymbol(SymbolName(key, keyLength), STUnknown);

if (symRef)

{

\*symRef = sym;

}

}

else

{

Assert(symRef);

if (\*symRef)

{

sym = \*symRef;

}

else

{

this->AssignPropertyId(pid);

return nullptr;

}

key = reinterpret\_cast<const wchar\_t\*>(sym->GetPid()->Psz());

}

Scope \*symScope = sym->GetScope();

Assert(symScope);

#if DBG\_DUMP

if (this->Trace())

{

if (sym != nullptr)

{

Output::Print(L"resolved %s to symbol of type %s: \n", key, sym->GetSymbolTypeName());

}

else

{

Output::Print(L"did not resolve %s\n", key);

}

}

#endif

if (!(sym->GetIsGlobal()))

{

FuncInfo \*top = funcInfoStack->Top();

bool nonLocalRef = symScope->GetFunc() != top;

if (forReference)

{

Js::PropertyId i;

Scope \*scope = FindScopeForSym(symScope, nullptr, &i, top);

// If we have a reference to a local within a with, we want to generate a closure represented by an object.

if (scope != symScope && scope->GetIsDynamic())

{

nonLocalRef = true;

symScope->SetIsObject();

}

}

if (nonLocalRef)

{

// Symbol referenced through a closure. Mark it as such and give it a property ID.

sym->SetHasNonLocalReference(true, this);

sym->SetPosition(top->byteCodeFunction->GetOrAddPropertyIdTracked(sym->GetName()));

// If this is var is local to a function (meaning that it belongs to the function's scope

// \*or\* to scope that need not be instantiated, like a function expression scope, which we'll

// merge with the function scope, then indicate that fact.

symScope->SetHasLocalInClosure(true);

if (symScope->GetFunc()->GetHasArguments() && sym->GetIsFormal())

{

// A formal is referenced non-locally. We need to allocate it on the heap, so

// do the same for the whole arguments object.

// Formal is referenced. So count of formals to function > 0.

// So no need to check for inParams here.

symScope->GetFunc()->SetHasHeapArguments(true);

}

if (symScope->GetFunc() != top)

{

top->SetHasClosureReference(true);

}

}

else if (sym->GetHasNonLocalReference() && !sym->GetIsCommittedToSlot() && !sym->HasVisitedCapturingFunc())

{

sym->SetHasNonCommittedReference(true);

}

if (sym->GetFuncExpr())

{

symScope->GetFunc()->SetFuncExprNameReference(true);

// If the func expr is captured by a closure, and if it's not getting its own scope,

// we have to make the function's scope a dynamic object. That's because we have to init

// the property as non-writable, and if we put it directly in the slots we can't protect it from being

// overwritten.

if (nonLocalRef && !symScope->GetFunc()->GetCallsEval() && !symScope->GetFunc()->GetChildCallsEval())

{

symScope->GetFunc()->GetParamScope()->SetIsObject();

symScope->GetFunc()->GetBodyScope()->SetIsObject();

}

}

}

return sym;

}

Symbol \* ByteCodeGenerator::AddSymbolToScope(Scope \*scope, const wchar\_t \*key, int keyLength, ParseNode \*varDecl, SymbolType symbolType)

{

Symbol \*sym = nullptr;

if (!UseParserBindings())

{

SymbolName const symName(key, keyLength);

if (scope->GetScopeType() == ScopeType\_FunctionBody)

{

sym = scope->GetFunc()->GetParamScope()->FindLocalSymbol(symName);

}

if (sym == nullptr)

{

sym = scope->FindLocalSymbol(symName);

}

if (sym == nullptr)

{

sym = Anew(alloc, Symbol, symName, varDecl, symbolType);

scope->AddNewSymbol(sym);

#if DBG\_DUMP

if (this->Trace())

{

Output::Print(L"added symbol %s of type %s to scope %x\n", key, sym->GetSymbolTypeName(), scope);

}

#endif

}

}

else

{

switch (varDecl->nop)

{

case knopConstDecl:

case knopLetDecl:

case knopVarDecl:

sym = varDecl->sxVar.sym/\*New\*/;

break;

case knopName:

AnalysisAssert(varDecl->sxPid.symRef);

sym = \*varDecl->sxPid.symRef;

break;

default:

AnalysisAssert(0);

sym = nullptr;

break;

}

if (sym->GetScope() != scope && sym->GetScope()->GetScopeType() != ScopeType\_Parameter)

{

// This can happen when we have a function declared at global eval scope, and it has

// references in deferred function bodies inside the eval. The BCG creates a new global scope

// on such compiles, so we essentially have to migrate the symbol to the new scope.

// We check fscrEvalCode, not fscrEval, because the same thing can happen in indirect eval,

// when fscrEval is not set.

Assert(((this->flags & fscrEvalCode) && sym->GetIsGlobal() && sym->GetSymbolType() == STFunction) || this->IsConsoleScopeEval());

Assert(scope->GetScopeType() == ScopeType\_Global);

scope->AddNewSymbol(sym);

}

}

Assert(sym && sym->GetScope() && (sym->GetScope() == scope || sym->GetScope()->GetScopeType() == ScopeType\_Parameter));

return sym;

}

Symbol \* ByteCodeGenerator::AddSymbolToFunctionScope(const wchar\_t \*key, int keyLength, ParseNode \*varDecl, SymbolType symbolType)

{

Scope\* scope = currentScope->GetFunc()->GetBodyScope();

return this->AddSymbolToScope(scope, key, keyLength, varDecl, symbolType);

}

FuncInfo \*ByteCodeGenerator::FindEnclosingNonLambda()

{

for (Scope \*scope = TopFuncInfo()->GetBodyScope(); scope; scope = scope->GetEnclosingScope())

{

if (!scope->GetFunc()->IsLambda())

{

return scope->GetFunc();

}

}

Assert(0);

return nullptr;

}

bool ByteCodeGenerator::CanStackNestedFunc(FuncInfo \* funcInfo, bool trace)

{

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

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

#endif

Assert(!funcInfo->IsGlobalFunction());

bool const doStackNestedFunc = !funcInfo->HasMaybeEscapedNestedFunc() && !IsInDebugMode() && !funcInfo->byteCodeFunction->IsGenerator();

if (!doStackNestedFunc)

{

return false;

}

bool callsEval = funcInfo->GetCallsEval() || funcInfo->GetChildCallsEval();

if (callsEval)

{

if (trace)

{

PHASE\_PRINT\_TESTTRACE(Js::StackFuncPhase, funcInfo->byteCodeFunction,

L"HasMaybeEscapedNestedFunc (Eval): %s (function %s)\n",

funcInfo->byteCodeFunction->GetDisplayName(),

funcInfo->byteCodeFunction->GetDebugNumberSet(debugStringBuffer));

}

return false;

}

if (funcInfo->GetBodyScope()->GetIsObject() || funcInfo->GetParamScope()->GetIsObject())

{

if (trace)

{

PHASE\_PRINT\_TESTTRACE(Js::StackFuncPhase, funcInfo->byteCodeFunction,

L"HasMaybeEscapedNestedFunc (ObjectScope): %s (function %s)\n",

funcInfo->byteCodeFunction->GetDisplayName(),

funcInfo->byteCodeFunction->GetDebugNumberSet(debugStringBuffer));

}

return false;

}

if (trace && funcInfo->byteCodeFunction->GetNestedCount())

{

// Only print functions that actually have nested functions, although we will still mark

// functions that don't have nested child functions as DoStackNestedFunc.

PHASE\_PRINT\_TESTTRACE(Js::StackFuncPhase, funcInfo->byteCodeFunction,

L"DoStackNestedFunc: %s (function %s)\n",

funcInfo->byteCodeFunction->GetDisplayName(),

funcInfo->byteCodeFunction->GetDebugNumberSet(debugStringBuffer));

}

return !PHASE\_OFF(Js::StackFuncPhase, funcInfo->byteCodeFunction);

}

bool ByteCodeGenerator::NeedObjectAsFunctionScope(FuncInfo \* funcInfo, ParseNode \* pnodeFnc) const

{

return funcInfo->GetCallsEval()

|| funcInfo->GetChildCallsEval()

|| NeedScopeObjectForArguments(funcInfo, pnodeFnc)

|| (this->flags & (fscrEval | fscrImplicitThis | fscrImplicitParents));

}

Scope \* ByteCodeGenerator::FindScopeForSym(Scope \*symScope, Scope \*scope, Js::PropertyId \*envIndex, FuncInfo \*funcInfo) const

{

for (scope = scope ? scope->GetEnclosingScope() : currentScope; scope; scope = scope->GetEnclosingScope())

{

if (scope->GetFunc() != funcInfo && scope->GetMustInstantiate() && scope != this->globalScope)

{

(\*envIndex)++;

}

if (scope == symScope || scope->GetIsDynamic())

{

break;

}

}

Assert(scope);

return scope;

}

/\* static \*/

Js::OpCode ByteCodeGenerator::GetStFldOpCode(FuncInfo\* funcInfo, bool isRoot, bool isLetDecl, bool isConstDecl, bool isClassMemberInit)

{

return GetStFldOpCode(funcInfo->GetIsStrictMode(), isRoot, isLetDecl, isConstDecl, isClassMemberInit);

}

/\* static \*/

Js::OpCode ByteCodeGenerator::GetScopedStFldOpCode(FuncInfo\* funcInfo, bool isConsoleScopeLetConst)

{

if (isConsoleScopeLetConst)

{

return Js::OpCode::ConsoleScopedStFld;

}

return GetScopedStFldOpCode(funcInfo->GetIsStrictMode());

}

/\* static \*/

Js::OpCode ByteCodeGenerator::GetStElemIOpCode(FuncInfo\* funcInfo)

{

return GetStElemIOpCode(funcInfo->GetIsStrictMode());

}

bool ByteCodeGenerator::DoJitLoopBodies(FuncInfo \*funcInfo) const

{

// Never JIT loop bodies in a function with a try.

// Otherwise, always JIT loop bodies under /forcejitloopbody.

// Otherwise, JIT loop bodies unless we're in eval/"new Function" or feature is disabled.

Assert(funcInfo->byteCodeFunction->IsFunctionParsed());

Js::FunctionBody\* functionBody = funcInfo->byteCodeFunction->GetFunctionBody();

return functionBody->ForceJITLoopBody() || funcInfo->byteCodeFunction->IsJitLoopBodyPhaseEnabled();

}

void ByteCodeGenerator::Generate(\_\_in ParseNode \*pnode, ulong grfscr, \_\_in ByteCodeGenerator\* byteCodeGenerator,

\_\_inout Js::ParseableFunctionInfo \*\* ppRootFunc, \_\_in uint sourceIndex,

\_\_in bool forceNoNative, \_\_in Parser\* parser, Js::ScriptFunction \*\*functionRef)

{

Js::ScriptContext \* scriptContext = byteCodeGenerator->scriptContext;

#ifdef PROFILE\_EXEC

scriptContext->ProfileBegin(Js::ByteCodePhase);

#endif

JS\_ETW(EventWriteJSCRIPT\_BYTECODEGEN\_START(scriptContext, 0));

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

Js::Utf8SourceInfo \* utf8SourceInfo = scriptContext->GetSource(sourceIndex);

byteCodeGenerator->m\_utf8SourceInfo = utf8SourceInfo;

// For dynamic code, just provide a small number since that source info should have very few functions

// For static code, the nextLocalFunctionId is a good guess of the initial size of the array to minimize reallocs

SourceContextInfo \* sourceContextInfo = utf8SourceInfo->GetSrcInfo()->sourceContextInfo;

utf8SourceInfo->EnsureInitialized((grfscr & fscrDynamicCode) ? 4 : (sourceContextInfo->nextLocalFunctionId - pnode->sxFnc.functionId));

sourceContextInfo->EnsureInitialized();

ArenaAllocator localAlloc(L"ByteCode", threadContext->GetPageAllocator(), Js::Throw::OutOfMemory);

byteCodeGenerator->parser = parser;

byteCodeGenerator->SetCurrentSourceIndex(sourceIndex);

byteCodeGenerator->Begin(&localAlloc, grfscr, \*ppRootFunc);

byteCodeGenerator->functionRef = functionRef;

Visit(pnode, byteCodeGenerator, Bind, AssignRegisters);

byteCodeGenerator->forceNoNative = forceNoNative;

byteCodeGenerator->EmitProgram(pnode);

if (byteCodeGenerator->flags & fscrEval)

{

// The eval caller's frame always escapes if eval refers to the caller's arguments.

byteCodeGenerator->GetRootFunc()->GetFunctionBody()->SetFuncEscapes(

byteCodeGenerator->funcEscapes || pnode->sxProg.m\_UsesArgumentsAtGlobal);

}

#ifdef IR\_VIEWER

if (grfscr & fscrIrDumpEnable)

{

byteCodeGenerator->GetRootFunc()->GetFunctionBody()->SetIRDumpEnabled(true);

}

#endif /\* IR\_VIEWER \*/

byteCodeGenerator->CheckDeferParseHasMaybeEscapedNestedFunc();

#ifdef PROFILE\_EXEC

scriptContext->ProfileEnd(Js::ByteCodePhase);

#endif

JS\_ETW(EventWriteJSCRIPT\_BYTECODEGEN\_STOP(scriptContext, 0));

#if ENABLE\_NATIVE\_CODEGEN && defined(ENABLE\_PREJIT)

if (!byteCodeGenerator->forceNoNative && !scriptContext->GetConfig()->IsNoNative()

&& Js::Configuration::Global.flags.Prejit

&& (grfscr & fscrNoPreJit) == 0)

{

GenerateAllFunctions(scriptContext->GetNativeCodeGenerator(), byteCodeGenerator->GetRootFunc()->GetFunctionBody());

}

#endif

if (ppRootFunc)

{

\*ppRootFunc = byteCodeGenerator->GetRootFunc();

}

#ifdef PERF\_COUNTERS

PHASE\_PRINT\_TESTTRACE1(Js::DeferParsePhase, L"TestTrace: deferparse - # of func: %d # deferparsed: %d\n",

PerfCounter::CodeCounterSet::GetTotalFunctionCounter().GetValue(), PerfCounter::CodeCounterSet::GetDeferedFunctionCounter().GetValue());

#endif

}

void ByteCodeGenerator::CheckDeferParseHasMaybeEscapedNestedFunc()

{

if (!this->parentScopeInfo)

{

return;

}

Assert(CONFIG\_FLAG(DeferNested));

Assert(this->funcInfoStack && !this->funcInfoStack->Empty());

// Box the stack nested function if we detected new may be escaped use function.

SList<FuncInfo \*>::Iterator i(this->funcInfoStack);

bool succeed = i.Next();

Assert(succeed);

Assert(i.Data()->IsGlobalFunction()); // We always leave a glo on type when defer parsing.

Assert(!i.Data()->IsRestored());

succeed = i.Next();

FuncInfo \* top = i.Data();

Assert(!top->IsGlobalFunction());

Assert(top->IsRestored());

Js::FunctionBody \* rootFuncBody = this->GetRootFunc()->GetFunctionBody();

if (!rootFuncBody->DoStackNestedFunc())

{

top->SetHasMaybeEscapedNestedFunc(DebugOnly(L"DeferredChild"));

}

else

{

// We have to wait until it is parsed before we populate the stack nested func parent.

Js::FunctionBody \* parentFunctionBody = nullptr;

FuncInfo \* parentFunc = top->GetBodyScope()->GetEnclosingFunc();

if (!parentFunc->IsGlobalFunction())

{

parentFunctionBody = parentFunc->GetParsedFunctionBody();

Assert(parentFunctionBody != rootFuncBody);

if (parentFunctionBody->DoStackNestedFunc())

{

rootFuncBody->SetStackNestedFuncParent(parentFunctionBody);

}

}

}

do

{

FuncInfo \* funcInfo = i.Data();

Assert(funcInfo->IsRestored());

Js::FunctionBody \* functionBody = funcInfo->GetParsedFunctionBody();

bool didStackNestedFunc = functionBody->DoStackNestedFunc();

if (!didStackNestedFunc)

{

return;

}

if (funcInfo->HasMaybeEscapedNestedFunc())

{

// This should box the rest of the parent functions.

if (PHASE\_TESTTRACE(Js::StackFuncPhase, this->pCurrentFunction))

{

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

Output::Print(L"DeferParse: box and disable stack function: %s (function %s)\n",

functionBody->GetDisplayName(), functionBody->GetDebugNumberSet(debugStringBuffer));

Output::Flush();

}

// During the box workflow we reset all the parents of all nested functions and up. If a fault occurs when the stack function

// is created this will cause further issues when trying to use the function object again. So failing faster seems to make more sense.

try

{

Js::StackScriptFunction::Box(functionBody, functionRef);

}

catch (Js::OutOfMemoryException)

{

FailedToBox\_OOM\_fatal\_error((ULONG\_PTR)functionBody);

}

return;

}

}

while (i.Next());

}

void ByteCodeGenerator::Begin(

\_\_in ArenaAllocator \*alloc,

\_\_in ulong grfscr,

\_\_in Js::ParseableFunctionInfo\* pRootFunc)

{

this->alloc = alloc;

this->flags = grfscr;

this->pRootFunc = pRootFunc;

this->pCurrentFunction = pRootFunc ? pRootFunc->GetFunctionBody() : nullptr;

if (this->pCurrentFunction && this->pCurrentFunction->GetIsGlobalFunc() && IsInNonDebugMode())

{

// This is the deferred parse case (not due to debug mode), in which case the global function will not be marked to compiled again.

this->pCurrentFunction = nullptr;

}

this->globalScope = nullptr;

this->currentScope = nullptr;

this->currentBlock = nullptr;

this->isBinding = true;

this->inPrologue = false;

this->funcEscapes = false;

this->maxAstSize = 0;

this->loopDepth = 0;

this->envDepth = 0;

this->trackEnvDepth = false;

this->funcInfoStack = Anew(alloc, SList<FuncInfo\*>, alloc);

// If pRootFunc is not null, this is a deferred parse function

// so reuse the property record list bound there since some of the symbols could have

// been bound. If it's null, we need to create a new property record list

if (pRootFunc != nullptr)

{

this->propertyRecords = pRootFunc->GetBoundPropertyRecords();

}

else

{

this->propertyRecords = nullptr;

}

Js::FunctionBody \*fakeGlobalFunc = scriptContext->GetFakeGlobalFuncForUndefer();

if (fakeGlobalFunc)

{

fakeGlobalFunc->ClearBoundPropertyRecords();

}

}

HRESULT GenerateByteCode(\_\_in ParseNode \*pnode, \_\_in ulong grfscr, \_\_in Js::ScriptContext\* scriptContext, \_\_inout Js::ParseableFunctionInfo \*\* ppRootFunc,

\_\_in uint sourceIndex, \_\_in bool forceNoNative, \_\_in Parser\* parser, \_\_in CompileScriptException \*pse, Js::ScopeInfo\* parentScopeInfo,

Js::ScriptFunction \*\* functionRef)

{

HRESULT hr = S\_OK;

ByteCodeGenerator byteCodeGenerator(scriptContext, parentScopeInfo);

BEGIN\_TRANSLATE\_EXCEPTION\_TO\_HRESULT\_NESTED

{

// Main code.

ByteCodeGenerator::Generate(pnode, grfscr, &byteCodeGenerator, ppRootFunc, sourceIndex, forceNoNative, parser, functionRef);

}

END\_TRANSLATE\_EXCEPTION\_TO\_HRESULT(hr);

if (FAILED(hr))

{

hr = pse->ProcessError(nullptr, hr, nullptr);

}

return hr;

}

void BindInstAndMember(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator)

{

Assert(pnode->nop == knopDot);

BindReference(pnode, byteCodeGenerator);

ParseNode \*right = pnode->sxBin.pnode2;

Assert(right->nop == knopName);

byteCodeGenerator->AssignPropertyId(right->sxPid.pid);

right->sxPid.sym = nullptr;

right->sxPid.symRef = nullptr;

right->grfpn |= fpnMemberReference;

}

void BindReference(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator)

{

// Do special reference-op binding so that we can, for instance, handle call from inside "with"

// where the "this" instance must be found dynamically.

bool isCallNode = false;

bool funcEscapes = false;

switch (pnode->nop)

{

case knopCall:

isCallNode = true;

pnode = pnode->sxCall.pnodeTarget;

break;

case knopDelete:

case knopTypeof:

pnode = pnode->sxUni.pnode1;

break;

case knopDot:

case knopIndex:

funcEscapes = true;

// fall through

case knopAsg:

pnode = pnode->sxBin.pnode1;

break;

default:

AssertMsg(0, "Unexpected opcode in BindReference");

return;

}

if (pnode->nop == knopName)

{

pnode->sxPid.sym = byteCodeGenerator->FindSymbol(pnode->sxPid.symRef, pnode->sxPid.pid, isCallNode);

if (funcEscapes &&

pnode->sxPid.sym &&

pnode->sxPid.sym->GetSymbolType() == STFunction &&

(!pnode->sxPid.sym->GetIsGlobal() || (byteCodeGenerator->GetFlags() & fscrEval)))

{

// Dot, index, and scope ops can cause a local function on the LHS to escape.

// Make sure scopes are not cached in this case.

byteCodeGenerator->FuncEscapes(pnode->sxPid.sym->GetScope());

}

}

}

void MarkFormal(ByteCodeGenerator \*byteCodeGenerator, Symbol \*formal, bool assignLocation, bool needDeclaration)

{

if (assignLocation)

{

formal->SetLocation(byteCodeGenerator->NextVarRegister());

}

if (needDeclaration)

{

formal->SetNeedDeclaration(true);

}

}

void AddArgsToScope(ParseNodePtr pnode, ByteCodeGenerator \*byteCodeGenerator, bool assignLocation)

{

Assert(byteCodeGenerator->TopFuncInfo()->varRegsCount == 0);

Js::ArgSlot pos = 1;

bool isSimpleParameterList = pnode->sxFnc.IsSimpleParameterList();

auto addArgToScope = [&](ParseNode \*arg)

{

if (arg->IsVarLetOrConst())

{

Symbol \*formal = byteCodeGenerator->AddSymbolToScope(byteCodeGenerator->TopFuncInfo()->GetParamScope(),

reinterpret\_cast<const wchar\_t\*>(arg->sxVar.pid->Psz()),

arg->sxVar.pid->Cch(),

arg,

STFormal);

#if DBG\_DUMP

if (byteCodeGenerator->Trace())

{

Output::Print(L"current context has declared arg %s of type %s at position %d\n", arg->sxVar.pid->Psz(), formal->GetSymbolTypeName(), pos);

}

#endif

if (!isSimpleParameterList)

{

formal->SetIsNonSimpleParameter(true);

}

arg->sxVar.sym = formal;

MarkFormal(byteCodeGenerator, formal, assignLocation || !isSimpleParameterList, !isSimpleParameterList);

}

else if (arg->nop == knopParamPattern)

{

arg->sxParamPattern.location = byteCodeGenerator->NextVarRegister();

}

else

{

Assert(false);

}

UInt16Math::Inc(pos);

};

// We process rest separately because the number of in args needs to exclude rest.

MapFormalsWithoutRest(pnode, addArgToScope);

byteCodeGenerator->SetNumberOfInArgs(pos);

MapFormalsFromPattern(pnode, addArgToScope);

if (pnode->sxFnc.pnodeRest != nullptr)

{

// The rest parameter will always be in a register, regardless of whether it is in a scope slot.

// We save the assignLocation value for the assert condition below.

bool assignLocationSave = assignLocation;

assignLocation = true;

addArgToScope(pnode->sxFnc.pnodeRest);

assignLocation = assignLocationSave;

}

Assert(!assignLocation || byteCodeGenerator->TopFuncInfo()->varRegsCount + 1 == pos);

}

void AddVarsToScope(ParseNode \*vars, ByteCodeGenerator \*byteCodeGenerator)

{

while (vars != nullptr)

{

Symbol \*sym = nullptr;

if (!byteCodeGenerator->UseParserBindings()

&& byteCodeGenerator->GetCurrentScope()->GetFunc()->GetParamScope() != nullptr)

{

SymbolName const symName(reinterpret\_cast<const wchar\_t\*>(vars->sxVar.pid->Psz()), vars->sxVar.pid->Cch());

// If we are not using parser bindings, we need to check that the sym is not in the parameter scope before adding it.

// Fetch the sym we just added from the param scope.

sym = byteCodeGenerator->GetCurrentScope()->GetFunc()->GetParamScope()->FindLocalSymbol(symName);

// Arguments needs to be created at parameter scope.

if (sym == nullptr && vars->grfpn & PNodeFlags::fpnArguments)

{

Scope\* scope = byteCodeGenerator->GetCurrentScope()->GetFunc()->GetParamScope();

sym = byteCodeGenerator->AddSymbolToScope(scope, reinterpret\_cast<const wchar\_t\*>(vars->sxVar.pid->Psz()), vars->sxVar.pid->Cch(), vars, STVariable);

}

}

if (sym == nullptr)

{

sym = byteCodeGenerator->AddSymbolToFunctionScope(reinterpret\_cast<const wchar\_t\*>(vars->sxVar.pid->Psz()), vars->sxVar.pid->Cch(), vars, STVariable);

}

#if DBG\_DUMP

if (sym->GetSymbolType() == STVariable && byteCodeGenerator->Trace())

{

Output::Print(L"current context has declared var %s of type %s\n",

vars->sxVar.pid->Psz(), sym->GetSymbolTypeName());

}

#endif

if (sym->GetIsArguments() || vars->sxVar.pnodeInit == nullptr)

{

// LHS's of var decls are usually bound to symbols later, during the Visit/Bind pass,

// so that things like catch scopes can be taken into account.

// The exception is "arguments", which always binds to the local scope.

// We can also bind to the function scope symbol now if there's no init value

// to assign.

vars->sxVar.sym = sym;

if (sym->GetIsArguments())

{

byteCodeGenerator->TopFuncInfo()->SetArgumentsSymbol(sym);

}

}

else

{

vars->sxVar.sym = nullptr;

}

vars = vars->sxVar.pnodeNext;

}

}

template <class Fn>

void VisitFncDecls(ParseNode \*fns, Fn action)

{

while (fns != nullptr)

{

switch (fns->nop)

{

case knopFncDecl:

action(fns);

fns = fns->sxFnc.pnodeNext;

break;

case knopBlock:

fns = fns->sxBlock.pnodeNext;

break;

case knopCatch:

fns = fns->sxCatch.pnodeNext;

break;

case knopWith:

fns = fns->sxWith.pnodeNext;

break;

default:

AssertMsg(false, "Unexpected opcode in tree of scopes");

return;

}

}

}

FuncInfo\* PreVisitFunction(ParseNode\* pnode, ByteCodeGenerator\* byteCodeGenerator)

{

// Do binding of function name(s), initialize function scope, propagate function-wide properties from

// the parent (if any).

FuncInfo\* parentFunc = byteCodeGenerator->TopFuncInfo();

// fIsRoot indicates that this is the root function to be returned to a ParseProcedureText/AddScriptLet/etc. call.

// In such cases, the global function is just a wrapper around the root function's declaration.

// We used to assert that this was the only top-level function body, but it's possible to trick

// "new Function" into compiling more than one function (see WOOB 1121759).

bool fIsRoot = (!(byteCodeGenerator->GetFlags() & fscrGlobalCode) &&

parentFunc->IsGlobalFunction() &&

parentFunc->root->sxFnc.GetTopLevelScope() == pnode);

const wchar\_t \*funcName = Js::Constants::AnonymousFunction;

uint funcNameLength = Js::Constants::AnonymousFunctionLength;

uint functionNameOffset = 0;

bool funcExprWithName = false;

if (pnode->sxFnc.hint != nullptr)

{

funcName = reinterpret\_cast<const wchar\_t\*>(pnode->sxFnc.hint);

funcNameLength = pnode->sxFnc.hintLength;

functionNameOffset = pnode->sxFnc.hintOffset;

Assert(funcNameLength != 0 || funcNameLength == (int)wcslen(funcName));

}

if (pnode->sxFnc.IsDeclaration() || pnode->sxFnc.IsMethod())

{

// Class members have the fully qualified name stored in 'hint', no need to replace it.

if (pnode->sxFnc.pid && !pnode->sxFnc.IsClassMember())

{

funcName = reinterpret\_cast<const wchar\_t\*>(pnode->sxFnc.pid->Psz());

funcNameLength = pnode->sxFnc.pid->Cch();

functionNameOffset = 0;

}

}

else if ((pnode->sxFnc.pnodeName != nullptr) &&

(pnode->sxFnc.pnodeName->nop == knopVarDecl))

{

funcName = reinterpret\_cast<const wchar\_t\*>(pnode->sxFnc.pnodeName->sxVar.pid->Psz());

funcNameLength = pnode->sxFnc.pnodeName->sxVar.pid->Cch();

functionNameOffset = 0;

//

// create the new scope for Function expression only in ES5 mode

//

funcExprWithName = true;

}

if (byteCodeGenerator->Trace())

{

Output::Print(L"function start %s\n", funcName);

}

Assert(pnode->sxFnc.funcInfo == nullptr);

FuncInfo\* funcInfo = pnode->sxFnc.funcInfo = byteCodeGenerator->StartBindFunction(funcName, funcNameLength, functionNameOffset, &funcExprWithName, pnode);

funcInfo->byteCodeFunction->SetIsNamedFunctionExpression(funcExprWithName);

funcInfo->byteCodeFunction->SetIsNameIdentifierRef(pnode->sxFnc.isNameIdentifierRef);

if (fIsRoot)

{

byteCodeGenerator->SetRootFuncInfo(funcInfo);

}

if (pnode->sxFnc.pnodeBody == nullptr)

{

// This is a deferred byte code gen, so we're done.

// Process the formal arguments, even if there's no AST for the body, to support Function.length.

if (byteCodeGenerator->UseParserBindings())

{

Js::ArgSlot pos = 1;

// We skip the rest parameter here because it is not counted towards the in arg count.

MapFormalsWithoutRest(pnode, [&](ParseNode \*pnode) { UInt16Math::Inc(pos); });

byteCodeGenerator->SetNumberOfInArgs(pos);

}

else

{

AddArgsToScope(pnode, byteCodeGenerator, false);

}

return funcInfo;

}

if (pnode->sxFnc.HasReferenceableBuiltInArguments())

{

// The parser identified that there is a way to reference the built-in 'arguments' variable from this function. So, we

// need to determine whether we need to create the variable or not. We need to create the variable iff:

if (pnode->sxFnc.CallsEval())

{

// 1. eval is called.

// 2. when the debugging is enabled, since user can seek arguments during breakpoint.

funcInfo->SetHasArguments(true);

funcInfo->SetHasHeapArguments(true);

if (funcInfo->inArgsCount == 0)

{

// If no formals to function, no need to create the propertyid array

byteCodeGenerator->AssignNullConstRegister();

}

}

else if (pnode->sxFnc.UsesArguments())

{

// 3. the function directly references an 'arguments' identifier

funcInfo->SetHasArguments(true);

if (pnode->sxFnc.HasHeapArguments())

{

funcInfo->SetHasHeapArguments(true, !pnode->sxFnc.IsGenerator() /\*= Optimize arguments in backend\*/);

if (funcInfo->inArgsCount == 0)

{

// If no formals to function, no need to create the propertyid array

byteCodeGenerator->AssignNullConstRegister();

}

}

}

}

Js::FunctionBody\* parentFunctionBody = parentFunc->GetParsedFunctionBody();

if (funcInfo->GetHasArguments() ||

parentFunctionBody->GetHasOrParentHasArguments())

{

// The JIT uses this info, for instance, to narrow kills of array operations

funcInfo->GetParsedFunctionBody()->SetHasOrParentHasArguments(true);

}

PreVisitBlock(pnode->sxFnc.pnodeScopes, byteCodeGenerator);

// If we have arguments, we are going to need locations if the function is in strict mode or we have a non-simple parameter list. This is because we will not create a scope object.

bool assignLocationForFormals = !(funcInfo->GetHasHeapArguments() && ByteCodeGenerator::NeedScopeObjectForArguments(funcInfo, funcInfo->root));

AddArgsToScope(pnode, byteCodeGenerator, assignLocationForFormals);

PreVisitBlock(pnode->sxFnc.pnodeBodyScope, byteCodeGenerator);

AddVarsToScope(pnode->sxFnc.pnodeVars, byteCodeGenerator);

return funcInfo;

}

void AssignFuncSymRegister(ParseNode \* pnode, ByteCodeGenerator \* byteCodeGenerator, FuncInfo \* callee)

{

// register to hold the allocated function (in enclosing sequence of global statements)

// TODO: Make the parser identify uses of function decls as RHS's of expressions.

// Currently they're all marked as used, so they all get permanent (non-temp) registers.

if (pnode->sxFnc.pnodeName == nullptr)

{

return;

}

Assert(pnode->sxFnc.pnodeName->nop == knopVarDecl);

Symbol \*sym = pnode->sxFnc.pnodeName->sxVar.sym;

if (sym)

{

if (!sym->GetIsGlobal() && !(callee->funcExprScope && callee->funcExprScope->GetIsObject()))

{

// If the func decl is used, we have to give the expression a register to protect against:

// x.x = function f() {...};

// x.y = function f() {...};

// If we let the value reside in the local slot for f, then both assignments will get the

// second definition.

if (!pnode->sxFnc.IsDeclaration())

{

// A named function expression's name belongs to the enclosing scope.

// In ES5 mode, it is visible only inside the inner function.

// Allocate a register for the 'name' symbol from an appropriate register namespace.

if (callee->GetFuncExprNameReference())

{

// This is a function expression with a name, but probably doesn't have a use within

// the function. If that is the case then allocate a register for LdFuncExpr inside the function

// we just finished post-visiting.

if (sym->GetLocation() == Js::Constants::NoRegister)

{

sym->SetLocation(callee->NextVarRegister());

}

}

}

else

{

// Function declaration

byteCodeGenerator->AssignRegister(sym);

pnode->location = sym->GetLocation();

Assert(byteCodeGenerator->GetCurrentScope()->GetFunc() == sym->GetScope()->GetFunc());

Symbol \* functionScopeVarSym = sym->GetFuncScopeVarSym();

if (functionScopeVarSym &&

!functionScopeVarSym->GetIsGlobal() &&

!functionScopeVarSym->IsInSlot(sym->GetScope()->GetFunc()))

{

Assert(byteCodeGenerator->GetScriptContext()->GetConfig()->IsBlockScopeEnabled());

byteCodeGenerator->AssignRegister(functionScopeVarSym);

}

}

}

else if (!pnode->sxFnc.IsDeclaration())

{

if (sym->GetLocation() == Js::Constants::NoRegister)

{

// Here, we are assigning a register for the LdFuncExpr instruction inside the function we just finished

// post-visiting. The symbol is given a register from the register pool for the function we just finished

// post-visiting, rather than from the parent function's register pool.

sym->SetLocation(callee->NextVarRegister());

}

}

}

}

FuncInfo\* PostVisitFunction(ParseNode\* pnode, ByteCodeGenerator\* byteCodeGenerator)

{

// Assign function-wide registers such as local frame object, closure environment, etc., based on

// observed attributes. Propagate attributes to the parent function (if any).

FuncInfo \*top = byteCodeGenerator->TopFuncInfo();

Symbol \*sym = pnode->sxFnc.GetFuncSymbol();

bool funcExprWithName = !top->IsGlobalFunction() && sym && sym->GetFuncExpr();

if (top->IsLambda())

{

if (byteCodeGenerator->FindEnclosingNonLambda()->isThisLexicallyCaptured)

{

top->byteCodeFunction->SetCapturesThis();

}

}

// If this is a named function expression and has deferred child, mark has non-local reference.

if (funcExprWithName)

{

// If we are reparsing this function due to being in debug mode - we should restore the state of this from the earlier parse

if (top->byteCodeFunction->IsFunctionParsed() && top->GetParsedFunctionBody()->HasFuncExprNameReference())

{

top->SetFuncExprNameReference(true);

sym->SetHasNonLocalReference(true, byteCodeGenerator);

top->SetHasLocalInClosure(true);

}

else if (top->HasDeferredChild())

{

// Before doing this, though, make sure there's no local symbol that hides the function name

// from the nested functions. If a lookup starting at the current local scope finds some symbol

// other than the func expr, then it's hidden. (See Win8 393618.)

Assert(CONFIG\_FLAG(DeferNested));

sym->SetHasNonLocalReference(true, byteCodeGenerator);

top->SetHasLocalInClosure(true);

if (pnode->sxFnc.pnodeBody)

{

top->GetParsedFunctionBody()->SetAllNonLocalReferenced(true);

}

if (!top->root->sxFnc.NameIsHidden())

{

top->SetFuncExprNameReference(true);

if (pnode->sxFnc.pnodeBody)

{

top->GetParsedFunctionBody()->SetFuncExprNameReference(true);

}

top->GetBodyScope()->SetIsObject();

top->GetParamScope()->SetIsObject();

if (pnode->sxFnc.pnodeBody)

{

top->GetParsedFunctionBody()->SetHasSetIsObject(true);

}

}

}

}

if (pnode->nop != knopProg

&& !top->bodyScope->GetIsObject()

&& byteCodeGenerator->NeedObjectAsFunctionScope(top, pnode))

{

// Even if it wasn't determined during visiting this function that we need a scope object, we still have a few conditions that may require one.

top->bodyScope->SetIsObject();

}

if (pnode->nop == knopProg

&& top->byteCodeFunction->GetIsStrictMode()

&& (byteCodeGenerator->GetFlags() & fscrEval))

{

// At global scope inside a strict mode eval, vars will not leak out and require a scope object (along with its parent.)

top->bodyScope->SetIsObject();

}

if (pnode->sxFnc.pnodeBody)

{

if (!top->IsGlobalFunction())

{

PostVisitBlock(pnode->sxFnc.pnodeBodyScope, byteCodeGenerator);

PostVisitBlock(pnode->sxFnc.pnodeScopes, byteCodeGenerator);

}

if ((byteCodeGenerator->GetFlags() & fscrEvalCode) && top->GetCallsEval())

{

// Must establish "this" in case nested eval refers to it.

top->GetParsedFunctionBody()->SetHasThis(true);

}

// This function refers to the closure environment if:

// 1. it has a child function (we'll pass the environment to the constructor when the child is created -

// even if it's not needed, it's as cheap as loading "null" from the library);

// 2. it calls eval (and will use the environment to construct the scope chain to pass to eval);

// 3. it refers to a local defined in a parent function;

// 4. it refers to a global and some parent calls eval (which might declare the "global" locally);

// 5. it refers to a global and we're in an event handler;

// 6. it refers to a global and the function was declared inside a "with";

// 7. it refers to a global and we're in an eval expression.

if (pnode->sxFnc.nestedCount != 0 ||

top->GetCallsEval() ||

top->GetHasClosureReference() ||

((top->GetHasGlobalRef() &&

(byteCodeGenerator->InDynamicScope() ||

(byteCodeGenerator->GetFlags() & (fscrImplicitThis | fscrImplicitParents | fscrEval))))))

{

byteCodeGenerator->SetNeedEnvRegister();

if (top->GetIsEventHandler())

{

byteCodeGenerator->AssignThisRegister();

}

}

// This function needs to construct a local frame on the heap if it is not the global function (even in eval) and:

// 1. it calls eval, which may refer to or declare any locals in this frame;

// 2. a child calls eval (which may refer to locals through a closure);

// 3. it uses non-strict mode "arguments", so the arguments have to be put in a closure;

// 4. it defines a local that is used by a child function (read from a closure).

// 5. it is a main function that's wrapped in a function expression scope but has locals used through

// a closure (used in forReference function call cases in a with for example).

if (!top->IsGlobalFunction())

{

if (top->GetCallsEval() ||

top->GetChildCallsEval() ||

(top->GetHasArguments() && ByteCodeGenerator::NeedScopeObjectForArguments(top, pnode) && pnode->sxFnc.pnodeArgs != nullptr) ||

top->GetHasLocalInClosure() ||

top->funcExprScope && top->funcExprScope->GetMustInstantiate())

{

if (!top->GetCallsEval())

{

byteCodeGenerator->AssignFrameSlotsRegister();

}

if (byteCodeGenerator->NeedObjectAsFunctionScope(top, top->root)

|| top->bodyScope->GetIsObject()

|| top->paramScope->GetIsObject())

{

byteCodeGenerator->AssignFrameObjRegister();

}

// The function also needs to construct a frame display if:

// 1. it calls eval;

// 2. it has a child function.

// 3. When has arguments and in debug mode. So that frame display be there along with frame object register.

if (top->GetCallsEval() ||

pnode->sxFnc.nestedCount != 0

|| (top->GetHasArguments()

&& (pnode->sxFnc.pnodeArgs != nullptr)

&& byteCodeGenerator->IsInDebugMode()))

{

byteCodeGenerator->SetNeedEnvRegister(); // This to ensure that Env should be there when the FrameDisplay register is there.

byteCodeGenerator->AssignFrameDisplayRegister();

if (top->GetIsEventHandler())

{

byteCodeGenerator->AssignThisRegister();

}

}

}

if (top->GetHasArguments())

{

Symbol \*argSym = top->GetArgumentsSymbol();

Assert(argSym);

if (argSym)

{

Assert(top->bodyScope->GetScopeSlotCount() == 0);

Assert(top->sameNameArgsPlaceHolderSlotCount == 0);

byteCodeGenerator->AssignRegister(argSym);

uint i = 0;

auto setArgScopeSlot = [&](ParseNode \*pnodeArg)

{

if (pnodeArg->IsVarLetOrConst())

{

Symbol\* sym = pnodeArg->sxVar.sym;

if (sym->GetScopeSlot() != Js::Constants::NoProperty)

{

top->sameNameArgsPlaceHolderSlotCount++; // Same name args appeared before

}

sym->SetScopeSlot(i);

i++;

}

};

// We don't need to process the rest parameter here because it may not need a scope slot.

if (ByteCodeGenerator::NeedScopeObjectForArguments(top, pnode))

{

MapFormalsWithoutRest(pnode, setArgScopeSlot);

MapFormalsFromPattern(pnode, setArgScopeSlot);

}

top->paramScope->SetScopeSlotCount(i);

Assert(top->GetHasHeapArguments());

if (ByteCodeGenerator::NeedScopeObjectForArguments(top, pnode)

&& pnode->sxFnc.IsSimpleParameterList())

{

top->byteCodeFunction->SetHasImplicitArgIns(false);

}

}

}

}

else

{

Assert(top->IsGlobalFunction());

// eval is called in strict mode

bool newScopeForEval = (top->byteCodeFunction->GetIsStrictMode() && (byteCodeGenerator->GetFlags() & fscrEval));

if (newScopeForEval)

{

byteCodeGenerator->SetNeedEnvRegister();

byteCodeGenerator->AssignFrameObjRegister();

byteCodeGenerator->AssignFrameDisplayRegister();

}

}

Assert(!funcExprWithName || sym);

if (funcExprWithName)

{

Assert(top->funcExprScope);

// If the func expr may be accessed via eval, force the func expr scope into an object.

if (top->GetCallsEval() || top->GetChildCallsEval())

{

top->funcExprScope->SetIsObject();

}

if (top->funcExprScope->GetIsObject())

{

top->funcExprScope->SetLocation(byteCodeGenerator->NextVarRegister());

}

}

}

byteCodeGenerator->EndBindFunction(funcExprWithName);

// If the "child" is the global function, we're done.

if (top->IsGlobalFunction())

{

return top;

}

FuncInfo\* const parentFunc = byteCodeGenerator->TopFuncInfo();

Js::FunctionBody \* parentFunctionBody = parentFunc->byteCodeFunction->GetFunctionBody();

Assert(parentFunctionBody != nullptr);

bool const hasAnyDeferredChild = top->HasDeferredChild() || top->IsDeferred();

bool setHasNonLocalReference = parentFunctionBody->HasAllNonLocalReferenced();

// If we have any deferred child, we need to instantiate the fake global block scope if it is not empty

if (parentFunc->IsGlobalFunction())

{

if (hasAnyDeferredChild && byteCodeGenerator->IsEvalWithBlockScopingNoParentScopeInfo())

{

Scope \* globalEvalBlockScope = parentFunc->GetGlobalEvalBlockScope();

if (globalEvalBlockScope->Count() != 0 || parentFunc->isThisLexicallyCaptured)

{

// We must instantiate the eval block scope if we have any deferred child

// and force all symbol to have non local reference so it will be captured

// by the scope.

globalEvalBlockScope->SetMustInstantiate(true);

globalEvalBlockScope->ForceAllSymbolNonLocalReference(byteCodeGenerator);

parentFunc->SetHasDeferredChild();

}

}

}

else

{

if (setHasNonLocalReference)

{

// All locals are already marked as non-locals-referenced. Mark the parent as well.

if (parentFunctionBody->HasSetIsObject())

{

// Updated the current function, as per the previous stored info.

parentFunc->GetBodyScope()->SetIsObject();

parentFunc->GetParamScope()->SetIsObject();

}

}

// Propagate "hasDeferredChild" attribute back to parent.

if (hasAnyDeferredChild)

{

Assert(CONFIG\_FLAG(DeferNested));

parentFunc->SetHasDeferredChild();

// Anything in parent may have non-local reference from deferredChild.

setHasNonLocalReference = true;

parentFunctionBody->SetAllNonLocalReferenced(true);

// If a deferred child has with, parent scopes may contain symbols called inside the with.

// Current implementation needs the symScope isObject.

if (top->ChildHasWith() || pnode->sxFnc.HasWithStmt())

{

parentFunc->SetChildHasWith();

parentFunc->GetBodyScope()->SetIsObject();

parentFunc->GetParamScope()->SetIsObject();

// Record this for future use in the no-refresh debugging.

parentFunctionBody->SetHasSetIsObject(true);

}

}

// Propagate HasMaybeEscapedNestedFunc

if (!byteCodeGenerator->CanStackNestedFunc(top, false) ||

byteCodeGenerator->NeedObjectAsFunctionScope(top, pnode))

{

parentFunc->SetHasMaybeEscapedNestedFunc(DebugOnly(L"Child"));

}

}

if (top->GetCallsEval() || top->GetChildCallsEval())

{

parentFunc->SetChildCallsEval(true);

ParseNode \*currentBlock = byteCodeGenerator->GetCurrentBlock();

if (currentBlock)

{

Assert(currentBlock->nop == knopBlock);

currentBlock->sxBlock.SetChildCallsEval(true);

}

parentFunc->SetHasHeapArguments(true);

setHasNonLocalReference = true;

parentFunctionBody->SetAllNonLocalReferenced(true);

Scope \* const funcExprScope = top->funcExprScope;

if (funcExprScope)

{

// If we have the body scope as an object, the outer function expression scope also needs to be an object to propagate the name.

funcExprScope->SetIsObject();

}

if (parentFunc->inArgsCount == 1)

{

// If no formals to function, no need to create the propertyid array

byteCodeGenerator->AssignNullConstRegister();

}

}

if (setHasNonLocalReference && !parentFunctionBody->HasDoneAllNonLocalReferenced())

{

parentFunc->GetBodyScope()->ForceAllSymbolNonLocalReference(byteCodeGenerator);

if (!parentFunc->IsGlobalFunction())

{

parentFunc->GetParamScope()->ForceAllSymbolNonLocalReference(byteCodeGenerator);

}

parentFunctionBody->SetHasDoneAllNonLocalReferenced(true);

}

if (top->HasSuperReference())

{

top->AssignSuperRegister();

}

if (top->HasDirectSuper())

{

top->AssignSuperCtorRegister();

}

if (top->IsClassConstructor())

{

if (top->IsBaseClassConstructor())

{

// Base class constructor may not explicitly reference new.target but we always need to have it in order to construct the 'this' object.

top->AssignNewTargetRegister();

// Also must have a register to slot the 'this' object into.

top->AssignThisRegister();

}

else

{

// Derived class constructors need to check undefined against explicit return statements.

top->AssignUndefinedConstRegister();

top->AssignNewTargetRegister();

if (top->GetCallsEval() || top->GetChildCallsEval())

{

top->AssignThisRegister();

top->SetIsThisLexicallyCaptured();

top->SetIsNewTargetLexicallyCaptured();

top->SetIsSuperLexicallyCaptured();

top->SetIsSuperCtorLexicallyCaptured();

top->SetHasLocalInClosure(true);

top->SetHasClosureReference(true);

top->SetHasCapturedThis();

}

}

}

AssignFuncSymRegister(pnode, byteCodeGenerator, top);

return top;

}

void MarkInit(ParseNode\* pnode)

{

if (pnode->nop == knopList)

{

do

{

MarkInit(pnode->sxBin.pnode1);

pnode = pnode->sxBin.pnode2;

}

while (pnode->nop == knopList);

MarkInit(pnode);

}

else

{

Symbol \*sym = nullptr;

ParseNode \*pnodeInit = nullptr;

if (pnode->nop == knopVarDecl)

{

sym = pnode->sxVar.sym;

pnodeInit = pnode->sxVar.pnodeInit;

}

else if (pnode->nop == knopAsg && pnode->sxBin.pnode1->nop == knopName)

{

sym = pnode->sxBin.pnode1->sxPid.sym;

pnodeInit = pnode->sxBin.pnode2;

}

if (sym && !sym->GetIsUsed() && pnodeInit)

{

sym->SetHasInit(true);

if (sym->HasVisitedCapturingFunc())

{

sym->SetHasNonCommittedReference(false);

}

}

}

}

void AddFunctionsToScope(ParseNodePtr scope, ByteCodeGenerator \* byteCodeGenerator)

{

VisitFncDecls(scope, [byteCodeGenerator](ParseNode \*fn)

{

ParseNode \*pnodeName = fn->sxFnc.pnodeName;

if (pnodeName && pnodeName->nop == knopVarDecl && fn->sxFnc.IsDeclaration())

{

const wchar\_t \*fnName = pnodeName->sxVar.pid->Psz();

if (byteCodeGenerator->Trace())

{

Output::Print(L"current context has declared function %s\n", fnName);

}

// In ES6, functions are scoped to the block, which will be the current scope.

// Pre-ES6, function declarations are scoped to the function body, so get that scope.

Symbol \*sym;

if (!byteCodeGenerator->GetCurrentScope()->IsGlobalEvalBlockScope())

{

sym = byteCodeGenerator->AddSymbolToScope(byteCodeGenerator->GetCurrentScope(), fnName, pnodeName->sxVar.pid->Cch(), pnodeName, STFunction);

}

else

{

sym = byteCodeGenerator->AddSymbolToFunctionScope(fnName, pnodeName->sxVar.pid->Cch(), pnodeName, STFunction);

}

pnodeName->sxVar.sym = sym;

if (sym->GetIsGlobal())

{

FuncInfo\* func = byteCodeGenerator->TopFuncInfo();

func->SetHasGlobalRef(true);

}

if (byteCodeGenerator->GetScriptContext()->GetConfig()->IsBlockScopeEnabled()

&& sym->GetScope() != sym->GetScope()->GetFunc()->GetBodyScope()

&& sym->GetScope() != sym->GetScope()->GetFunc()->GetParamScope())

{

sym->SetIsBlockVar(true);

}

}

});

}

template <class PrefixFn, class PostfixFn>

void VisitNestedScopes(ParseNode\* pnodeScopeList, ParseNode\* pnodeParent, ByteCodeGenerator\* byteCodeGenerator,

PrefixFn prefix, PostfixFn postfix, uint \*pIndex)

{

// Visit all scopes nested in this scope before visiting this function's statements. This way we have all the

// attributes of all the inner functions before we assign registers within this function.

// All the attributes we need to propagate downward should already be recorded by the parser.

// - call to "eval()"

// - nested in "with"

Js::ParseableFunctionInfo\* parentFunc = pnodeParent->sxFnc.funcInfo->byteCodeFunction;

ParseNode\* pnodeScope;

uint i = 0;

// Cache to restore it back once we come out of current function.

Js::FunctionBody \* pLastReuseFunc = byteCodeGenerator->pCurrentFunction;

for (pnodeScope = pnodeScopeList; pnodeScope;)

{

switch (pnodeScope->nop)

{

case knopFncDecl:

if (pLastReuseFunc)

{

if (!byteCodeGenerator->IsInNonDebugMode())

{

// Here we are trying to match the inner sub-tree as well with already created inner function.

if ((pLastReuseFunc->GetIsGlobalFunc() && parentFunc->GetIsGlobalFunc())

|| (!pLastReuseFunc->GetIsGlobalFunc() && !parentFunc->GetIsGlobalFunc()))

{

Assert(pLastReuseFunc->StartInDocument() == pnodeParent->ichMin);

Assert(pLastReuseFunc->LengthInChars() == pnodeParent->LengthInCodepoints());

Assert(pLastReuseFunc->GetNestedCount() == parentFunc->GetNestedCount());

// If the current function is not parsed yet, its function body is not generated yet.

// Reset pCurrentFunction to null so that it will not be able re-use anything.

Js::FunctionProxy\* proxy = pLastReuseFunc->GetNestedFunc((\*pIndex));

if (proxy && proxy->IsFunctionBody())

{

byteCodeGenerator->pCurrentFunction = proxy->GetFunctionBody();

}

else

{

byteCodeGenerator->pCurrentFunction = nullptr;

}

}

}

else if (!parentFunc->GetIsGlobalFunc())

{

// In the deferred parsing mode, we will be reusing the only one function (which is asked when on ::Begin) all inner function will be created.

byteCodeGenerator->pCurrentFunction = nullptr;

}

}

PreVisitFunction(pnodeScope, byteCodeGenerator);

pnodeParent->sxFnc.funcInfo->OnStartVisitFunction(pnodeScope);

if (pnodeScope->sxFnc.pnodeBody)

{

if (!byteCodeGenerator->IsInNonDebugMode() && pLastReuseFunc != nullptr && byteCodeGenerator->pCurrentFunction == nullptr)

{

// Patch current non-parsed function's FunctionBodyImpl with the new generated function body.

// So that the function object (pointing to the old function body) can able to get to the new one.

Js::FunctionProxy\* proxy = pLastReuseFunc->GetNestedFunc((\*pIndex));

if (proxy && !proxy->IsFunctionBody())

{

proxy->UpdateFunctionBodyImpl(pnodeScope->sxFnc.funcInfo->byteCodeFunction->GetFunctionBody());

}

}

BeginVisitBlock(pnodeScope->sxFnc.pnodeScopes, byteCodeGenerator);

i = 0;

ParseNodePtr containerScope = pnodeScope->sxFnc.pnodeScopes;

VisitNestedScopes(containerScope, pnodeScope, byteCodeGenerator, prefix, postfix, &i);

MapFormals(pnodeScope, [&](ParseNode \*argNode) { Visit(argNode, byteCodeGenerator, prefix, postfix); });

if (!pnodeScope->sxFnc.IsSimpleParameterList())

{

byteCodeGenerator->AssignUndefinedConstRegister();

}

BeginVisitBlock(pnodeScope->sxFnc.pnodeBodyScope, byteCodeGenerator);

ParseNode\* pnode = pnodeScope->sxFnc.pnodeBody;

while (pnode->nop == knopList)

{

// Check to see whether initializations of locals to "undef" can be skipped.

// The logic to do this is cheap - omit the init if we see an init with a value

// on the RHS at the top statement level (i.e., not inside a block, try, loop, etc.)

// before we see a use. The motivation is to help identify single-def locals in the BE.

// Note that this can't be done for globals.

byteCodeGenerator->SetCurrentTopStatement(pnode->sxBin.pnode1);

Visit(pnode->sxBin.pnode1, byteCodeGenerator, prefix, postfix);

if (!pnodeScope->sxFnc.funcInfo->GetCallsEval() &&

!pnodeScope->sxFnc.funcInfo->GetChildCallsEval() &&

// So that it will not be marked as init thus it will be added to the diagnostics symbols container.

!(byteCodeGenerator->ShouldTrackDebuggerMetadata()))

{

MarkInit(pnode->sxBin.pnode1);

}

pnode = pnode->sxBin.pnode2;

}

byteCodeGenerator->SetCurrentTopStatement(pnode);

Visit(pnode, byteCodeGenerator, prefix, postfix);

EndVisitBlock(pnodeScope->sxFnc.pnodeBodyScope, byteCodeGenerator);

EndVisitBlock(pnodeScope->sxFnc.pnodeScopes, byteCodeGenerator);

}

else if (pnodeScope->sxFnc.nestedCount)

{

// The nested function is deferred but has its own nested functions.

// Make sure we at least zero-initialize its array in case, for instance, we get cloned

// before the function is called and the array filled in.

#ifdef RECYCLER\_WRITE\_BARRIER

WriteBarrierPtr<Js::FunctionProxy>::ClearArray(pnodeScope->sxFnc.funcInfo->byteCodeFunction->GetNestedFuncArray(), pnodeScope->sxFnc.nestedCount);

#else

memset(pnodeScope->sxFnc.funcInfo->byteCodeFunction->GetNestedFuncArray(), 0, pnodeScope->sxFnc.nestedCount \* sizeof(Js::FunctionBody\*));

#endif

}

pnodeScope->sxFnc.nestedIndex = \*pIndex;

parentFunc->SetNestedFunc(pnodeScope->sxFnc.funcInfo->byteCodeFunction, (\*pIndex)++, byteCodeGenerator->GetFlags());

Assert(parentFunc);

pnodeParent->sxFnc.funcInfo->OnEndVisitFunction(pnodeScope);

PostVisitFunction(pnodeScope, byteCodeGenerator);

// Merge parameter and body scopes, unless we are deferring the function.

// If we are deferring the function, we will need both scopes to do the proper binding when

// the function is undeferred. After the function is undeferred, it is safe to merge the scopes.

if (pnodeScope->sxFnc.funcInfo->paramScope != nullptr && pnodeScope->sxFnc.pnodeBody != nullptr)

{

Scope::MergeParamAndBodyScopes(pnodeScope, byteCodeGenerator);

}

pnodeScope = pnodeScope->sxFnc.pnodeNext;

byteCodeGenerator->pCurrentFunction = pLastReuseFunc;

break;

case knopBlock:

PreVisitBlock(pnodeScope, byteCodeGenerator);

pnodeParent->sxFnc.funcInfo->OnStartVisitScope(pnodeScope->sxBlock.scope);

VisitNestedScopes(pnodeScope->sxBlock.pnodeScopes, pnodeParent, byteCodeGenerator, prefix, postfix, pIndex);

pnodeParent->sxFnc.funcInfo->OnEndVisitScope(pnodeScope->sxBlock.scope);

PostVisitBlock(pnodeScope, byteCodeGenerator);

pnodeScope = pnodeScope->sxBlock.pnodeNext;

break;

case knopCatch:

PreVisitCatch(pnodeScope, byteCodeGenerator);

Visit(pnodeScope->sxCatch.pnodeParam, byteCodeGenerator, prefix, postfix);

if (pnodeScope->sxCatch.pnodeParam->nop == knopParamPattern)

{

if (pnodeScope->sxCatch.pnodeParam->sxParamPattern.location == Js::Constants::NoRegister)

{

pnodeScope->sxCatch.pnodeParam->sxParamPattern.location = byteCodeGenerator->NextVarRegister();

}

}

pnodeParent->sxFnc.funcInfo->OnStartVisitScope(pnodeScope->sxCatch.scope);

VisitNestedScopes(pnodeScope->sxCatch.pnodeScopes, pnodeParent, byteCodeGenerator, prefix, postfix, pIndex);

pnodeParent->sxFnc.funcInfo->OnEndVisitScope(pnodeScope->sxCatch.scope);

PostVisitCatch(pnodeScope, byteCodeGenerator);

pnodeScope = pnodeScope->sxCatch.pnodeNext;

break;

case knopWith:

PreVisitWith(pnodeScope, byteCodeGenerator);

pnodeParent->sxFnc.funcInfo->OnStartVisitScope(pnodeScope->sxWith.scope);

VisitNestedScopes(pnodeScope->sxWith.pnodeScopes, pnodeParent, byteCodeGenerator, prefix, postfix, pIndex);

pnodeParent->sxFnc.funcInfo->OnEndVisitScope(pnodeScope->sxWith.scope);

PostVisitWith(pnodeScope, byteCodeGenerator);

pnodeScope = pnodeScope->sxWith.pnodeNext;

break;

default:

AssertMsg(false, "Unexpected opcode in tree of scopes");

return;

}

}

}

void PreVisitBlock(ParseNode \*pnodeBlock, ByteCodeGenerator \*byteCodeGenerator)

{

if (!pnodeBlock->sxBlock.scope &&

!pnodeBlock->sxBlock.HasBlockScopedContent() &&

!pnodeBlock->sxBlock.GetCallsEval())

{

// Do nothing here if the block doesn't declare anything or call eval (which may declare something).

return;

}

bool isGlobalEvalBlockScope = false;

FuncInfo \*func = byteCodeGenerator->TopFuncInfo();

if (func->IsGlobalFunction() &&

func->root->sxFnc.pnodeScopes == pnodeBlock &&

byteCodeGenerator->IsEvalWithBlockScopingNoParentScopeInfo())

{

isGlobalEvalBlockScope = true;

}

Assert(!byteCodeGenerator->UseParserBindings() ||

!pnodeBlock->sxBlock.scope ||

isGlobalEvalBlockScope == (pnodeBlock->sxBlock.scope->GetScopeType() == ScopeType\_GlobalEvalBlock));

ArenaAllocator \*alloc = byteCodeGenerator->GetAllocator();

Scope \*scope;

if ((pnodeBlock->sxBlock.blockType == PnodeBlockType::Global && !byteCodeGenerator->IsEvalWithBlockScopingNoParentScopeInfo()) || pnodeBlock->sxBlock.blockType == PnodeBlockType::Function)

{

scope = byteCodeGenerator->GetCurrentScope();

if (pnodeBlock->sxBlock.blockType == PnodeBlockType::Function)

{

AnalysisAssert(pnodeBlock->sxBlock.scope);

if (pnodeBlock->sxBlock.scope->GetScopeType() == ScopeType\_Parameter

&& scope->GetScopeType() == ScopeType\_FunctionBody)

{

scope = scope->GetEnclosingScope();

}

}

pnodeBlock->sxBlock.scope = scope;

}

else if (!(pnodeBlock->grfpn & fpnSyntheticNode) || isGlobalEvalBlockScope)

{

Assert(byteCodeGenerator->GetScriptContext()->GetConfig()->IsBlockScopeEnabled());

scope = pnodeBlock->sxBlock.scope;

if (!scope || !byteCodeGenerator->UseParserBindings())

{

scope = Anew(alloc, Scope, alloc,

isGlobalEvalBlockScope? ScopeType\_GlobalEvalBlock : ScopeType\_Block, true);

pnodeBlock->sxBlock.scope = scope;

}

scope->SetFunc(byteCodeGenerator->TopFuncInfo());

// For now, prevent block scope from being merged with enclosing function scope.

// Consider optimizing this.

scope->SetCanMerge(false);

if (isGlobalEvalBlockScope)

{

scope->SetIsObject();

}

byteCodeGenerator->PushScope(scope);

byteCodeGenerator->PushBlock(pnodeBlock);

}

else

{

return;

}

Assert(scope && scope == pnodeBlock->sxBlock.scope);

bool isGlobalScope = (scope->GetEnclosingScope() == nullptr);

Assert(!isGlobalScope || (pnodeBlock->grfpn & fpnSyntheticNode));

// If it is the global eval block scope, we don't what function decl to be assigned in the block scope.

// They should already declared in the global function's scope.

if (!isGlobalEvalBlockScope && !isGlobalScope)

{

AddFunctionsToScope(pnodeBlock->sxBlock.pnodeScopes, byteCodeGenerator);

}

// We can skip this check by not creating the GlobalEvalBlock above and in Parser::Parse for console eval but that seems to break couple of places

// as we heavily depend on BlockHasOwnScope function. Once we clean up the creation of GlobalEvalBlock for evals we can clean this as well.

if (byteCodeGenerator->IsConsoleScopeEval() && isGlobalEvalBlockScope && !isGlobalScope)

{

AssertMsg(scope->GetEnclosingScope()->GetScopeType() == ScopeType\_Global, "Additional scope between Global and GlobalEvalBlock?");

scope = scope->GetEnclosingScope();

isGlobalScope = true;

}

auto addSymbolToScope = [scope, byteCodeGenerator, isGlobalScope](ParseNode \*pnode)

{

Symbol \*sym = byteCodeGenerator->AddSymbolToScope(scope, reinterpret\_cast<const wchar\_t\*>(pnode->sxVar.pid->Psz()), pnode->sxVar.pid->Cch(), pnode, STVariable);

#if DBG\_DUMP

if (sym->GetSymbolType() == STVariable && byteCodeGenerator->Trace())

{

Output::Print(L"current context has declared %s %s of type %s\n",

sym->GetDecl()->nop == knopLetDecl ? L"let" : L"const",

pnode->sxVar.pid->Psz(),

sym->GetSymbolTypeName());

}

#endif

sym->SetIsGlobal(isGlobalScope);

sym->SetIsBlockVar(true);

sym->SetNeedDeclaration(true);

pnode->sxVar.sym = sym;

};

byteCodeGenerator->IterateBlockScopedVariables(pnodeBlock, addSymbolToScope);

}

void PostVisitBlock(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator)

{

if (!BlockHasOwnScope(pnode, byteCodeGenerator))

{

return;

}

if (pnode->sxBlock.GetCallsEval() || pnode->sxBlock.GetChildCallsEval() || (byteCodeGenerator->GetFlags() & (fscrEval | fscrImplicitThis | fscrImplicitParents)))

{

Scope \*scope = pnode->sxBlock.scope;

bool scopeIsEmpty = scope->IsEmpty();

scope->SetIsObject();

scope->SetCapturesAll(true);

scope->SetMustInstantiate(!scopeIsEmpty);

}

byteCodeGenerator->PopScope();

byteCodeGenerator->PopBlock();

ParseNode \*currentBlock = byteCodeGenerator->GetCurrentBlock();

if (currentBlock && (pnode->sxBlock.GetCallsEval() || pnode->sxBlock.GetChildCallsEval()))

{

currentBlock->sxBlock.SetChildCallsEval(true);

}

}

void PreVisitCatch(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator)

{

// Push the catch scope and add the catch expression to it.

byteCodeGenerator->StartBindCatch(pnode);

if (pnode->sxCatch.pnodeParam->nop == knopParamPattern)

{

Parser::MapBindIdentifier(pnode->sxCatch.pnodeParam->sxParamPattern.pnode1, [&](ParseNodePtr item)

{

Symbol \*sym = item->sxVar.sym;

if (!byteCodeGenerator->UseParserBindings())

{

sym = byteCodeGenerator->AddSymbolToScope(pnode->sxCatch.scope, reinterpret\_cast<const wchar\_t\*>(item->sxVar.pid->Psz()), item->sxVar.pid->Cch(), item, STVariable);

item->sxVar.sym = sym;

}

#if DBG\_DUMP

if (byteCodeGenerator->Trace())

{

Output::Print(L"current context has declared catch var %s of type %s\n",

item->sxVar.pid->Psz(), sym->GetSymbolTypeName());

}

#endif

});

}

else

{

Symbol \*sym;

if (!byteCodeGenerator->UseParserBindings())

{

sym = byteCodeGenerator->AddSymbolToScope(pnode->sxCatch.scope, reinterpret\_cast<const wchar\_t\*>(pnode->sxCatch.pnodeParam->sxPid.pid->Psz()), pnode->sxCatch.pnodeParam->sxPid.pid->Cch(), pnode->sxCatch.pnodeParam, STVariable);

}

else

{

sym = \*pnode->sxCatch.pnodeParam->sxPid.symRef;

}

Assert(sym->GetScope() == pnode->sxCatch.scope);

#if DBG\_DUMP

if (byteCodeGenerator->Trace())

{

Output::Print(L"current context has declared catch var %s of type %s\n",

pnode->sxCatch.pnodeParam->sxPid.pid->Psz(), sym->GetSymbolTypeName());

}

#endif

sym->SetIsCatch(true);

pnode->sxCatch.pnodeParam->sxPid.sym = sym;

}

// This call will actually add the nested function symbols to the enclosing function scope (which is what we want).

AddFunctionsToScope(pnode->sxCatch.pnodeScopes, byteCodeGenerator);

}

void PostVisitCatch(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator)

{

byteCodeGenerator->EndBindCatch();

}

void PreVisitWith(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator)

{

ArenaAllocator \*alloc = byteCodeGenerator->GetAllocator();

Scope \*scope = Anew(alloc, Scope, alloc, ScopeType\_With);

scope->SetFunc(byteCodeGenerator->TopFuncInfo());

scope->SetIsDynamic(true);

pnode->sxWith.scope = scope;

byteCodeGenerator->PushScope(scope);

}

void PostVisitWith(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator)

{

byteCodeGenerator->PopScope();

}

void BindFuncSymbol(ParseNode \*pnodeFnc, ByteCodeGenerator \*byteCodeGenerator)

{

if (pnodeFnc->sxFnc.pnodeName)

{

Assert(pnodeFnc->sxFnc.pnodeName->nop == knopVarDecl);

Symbol \*sym = pnodeFnc->sxFnc.pnodeName->sxVar.sym;

FuncInfo\* func = byteCodeGenerator->TopFuncInfo();

if (sym == nullptr || sym->GetIsGlobal())

{

func->SetHasGlobalRef(true);

}

}

}

bool IsMathLibraryId(Js::PropertyId propertyId)

{

return (propertyId >= Js::PropertyIds::abs) && (propertyId <= Js::PropertyIds::fround);

}

bool IsLibraryFunction(ParseNode\* expr, Js::ScriptContext\* scriptContext)

{

if (expr && expr->nop == knopDot)

{

ParseNode\* lhs = expr->sxBin.pnode1;

ParseNode\* rhs = expr->sxBin.pnode2;

if ((lhs != nullptr) && (rhs != nullptr) && (lhs->nop == knopName) && (rhs->nop == knopName))

{

Symbol\* lsym = lhs->sxPid.sym;

if ((lsym == nullptr || lsym->GetIsGlobal()) && lhs->sxPid.PropertyIdFromNameNode() == Js::PropertyIds::Math)

{

return IsMathLibraryId(rhs->sxPid.PropertyIdFromNameNode());

}

}

}

return false;

}

struct SymCheck

{

static const int kMaxInvertedSyms = 8;

Symbol\* syms[kMaxInvertedSyms];

Symbol\* permittedSym;

int symCount;

bool result;

bool cond;

bool AddSymbol(Symbol\* sym)

{

if (symCount < kMaxInvertedSyms)

{

syms[symCount++] = sym;

return true;

}

else

{

return false;

}

}

bool MatchSymbol(Symbol\* sym)

{

if (sym != permittedSym)

{

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

{

if (sym == syms[i])

{

return true;

}

}

}

return false;

}

void Init()

{

symCount = 0;

result = true;

}

};

void CheckInvertableExpr(ParseNode\* pnode, ByteCodeGenerator\* byteCodeGenerator, SymCheck\* symCheck)

{

if (symCheck->result)

{

switch (pnode->nop)

{

case knopName:

if (symCheck->MatchSymbol(pnode->sxPid.sym))

{

symCheck->result = false;

}

break;

case knopCall:

{

ParseNode\* callTarget = pnode->sxBin.pnode1;

if (callTarget != nullptr)

{

if (callTarget->nop == knopName)

{

Symbol\* sym = callTarget->sxPid.sym;

if (sym && sym->SingleDef())

{

ParseNode\* decl = sym->GetDecl();

if (decl == nullptr ||

decl->nop != knopVarDecl ||

!IsLibraryFunction(decl->sxVar.pnodeInit, byteCodeGenerator->GetScriptContext()))

{

symCheck->result = false;

}

}

else

{

symCheck->result = false;

}

}

else if (callTarget->nop == knopDot)

{

if (!IsLibraryFunction(callTarget, byteCodeGenerator->GetScriptContext()))

{

symCheck->result = false;

}

}

}

else

{

symCheck->result = false;

}

break;

}

case knopDot:

if (!IsLibraryFunction(pnode, byteCodeGenerator->GetScriptContext()))

{

symCheck->result = false;

}

break;

case knopTrue:

case knopFalse:

case knopAdd:

case knopSub:

case knopDiv:

case knopMul:

case knopExpo:

case knopMod:

case knopNeg:

case knopInt:

case knopFlt:

case knopLt:

case knopGt:

case knopLe:

case knopGe:

case knopEq:

case knopNe:

break;

default:

symCheck->result = false;

break;

}

}

}

bool InvertableExpr(SymCheck\* symCheck, ParseNode\* expr, ByteCodeGenerator\* byteCodeGenerator)

{

symCheck->result = true;

symCheck->cond = false;

symCheck->permittedSym = nullptr;

VisitIndirect<SymCheck>(expr, byteCodeGenerator, symCheck, &CheckInvertableExpr, nullptr);

return symCheck->result;

}

bool InvertableExprPlus(SymCheck\* symCheck, ParseNode\* expr, ByteCodeGenerator\* byteCodeGenerator, Symbol\* permittedSym)

{

symCheck->result = true;

symCheck->cond = true;

symCheck->permittedSym = permittedSym;

VisitIndirect<SymCheck>(expr, byteCodeGenerator, symCheck, &CheckInvertableExpr, nullptr);

return symCheck->result;

}

void CheckLocalVarDef(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator)

{

Assert(pnode->nop == knopAsg);

if (pnode->sxBin.pnode1 != nullptr)

{

ParseNode \*lhs = pnode->sxBin.pnode1;

if (lhs->nop == knopName)

{

Symbol \*sym = lhs->sxPid.sym;

if (sym != nullptr)

{

sym->RecordDef();

}

}

}

}

ParseNode\* ConstructInvertedStatement(ParseNode\* stmt, ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* funcInfo,

ParseNode\*\* outerStmtRef)

{

if (stmt == nullptr)

{

return nullptr;

}

ParseNode\* cStmt;

if ((stmt->nop == knopAsg) || (stmt->nop == knopVarDecl))

{

ParseNode\* rhs = nullptr;

ParseNode\* lhs = nullptr;

if (stmt->nop == knopAsg)

{

rhs = stmt->sxBin.pnode2;

lhs = stmt->sxBin.pnode1;

}

else if (stmt->nop == knopVarDecl)

{

rhs = stmt->sxVar.pnodeInit;

}

ArenaAllocator\* alloc = byteCodeGenerator->GetAllocator();

ParseNode\* loopInvar = byteCodeGenerator->GetParser()->CreateTempNode(rhs);

loopInvar->location = funcInfo->NextVarRegister();

// Can't use a temp register here because the inversion happens at the parse tree level without generating

// any bytecode yet. All local non-temp registers need to be initialized for jitted loop bodies, and since this is

// not a user variable, track this register separately to have it be initialized at the top of the function.

funcInfo->nonUserNonTempRegistersToInitialize.Add(loopInvar->location);

// add temp node to list of initializers for new outer loop

if ((\*outerStmtRef)->sxBin.pnode1 == nullptr)

{

(\*outerStmtRef)->sxBin.pnode1 = loopInvar;

}

else

{

ParseNode\* listNode = Parser::StaticCreateBinNode(knopList, nullptr, nullptr, alloc);

(\*outerStmtRef)->sxBin.pnode2 = listNode;

listNode->sxBin.pnode1 = loopInvar;

\*outerStmtRef = listNode;

}

ParseNode\* tempName = byteCodeGenerator->GetParser()->CreateTempRef(loopInvar);

if (lhs != nullptr)

{

cStmt = Parser::StaticCreateBinNode(knopAsg, lhs, tempName, alloc);

}

else

{

// Use AddVarDeclNode to add the var to the function.

// Do not use CreateVarDeclNode which is meant to be used while parsing. It assumes that

// parser's internal data structures (m\_ppnodeVar in particular) is at the "current" location.

cStmt = byteCodeGenerator->GetParser()->AddVarDeclNode(stmt->sxVar.pid, funcInfo->root);

cStmt->sxVar.pnodeInit = tempName;

cStmt->sxVar.sym = stmt->sxVar.sym;

}

}

else

{

cStmt = byteCodeGenerator->GetParser()->CopyPnode(stmt);

}

return cStmt;

}

ParseNode\* ConstructInvertedLoop(ParseNode\* innerLoop, ParseNode\* outerLoop, ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* funcInfo)

{

ArenaAllocator\* alloc = byteCodeGenerator->GetAllocator();

ParseNode\* outerLoopC = Parser::StaticCreateNodeT<knopFor>(alloc);

outerLoopC->sxFor.pnodeInit = innerLoop->sxFor.pnodeInit;

outerLoopC->sxFor.pnodeCond = innerLoop->sxFor.pnodeCond;

outerLoopC->sxFor.pnodeIncr = innerLoop->sxFor.pnodeIncr;

outerLoopC->sxFor.pnodeBlock = innerLoop->sxFor.pnodeBlock;

outerLoopC->sxFor.pnodeInverted = nullptr;

ParseNode\* innerLoopC = Parser::StaticCreateNodeT<knopFor>(alloc);

innerLoopC->sxFor.pnodeInit = outerLoop->sxFor.pnodeInit;

innerLoopC->sxFor.pnodeCond = outerLoop->sxFor.pnodeCond;

innerLoopC->sxFor.pnodeIncr = outerLoop->sxFor.pnodeIncr;

innerLoopC->sxFor.pnodeBlock = outerLoop->sxFor.pnodeBlock;

innerLoopC->sxFor.pnodeInverted = nullptr;

ParseNode\* innerBod = Parser::StaticCreateBlockNode(alloc);

innerLoopC->sxFor.pnodeBody = innerBod;

innerBod->sxBlock.scope = innerLoop->sxFor.pnodeBody->sxBlock.scope;

ParseNode\* outerBod = Parser::StaticCreateBlockNode(alloc);

outerLoopC->sxFor.pnodeBody = outerBod;

outerBod->sxBlock.scope = outerLoop->sxFor.pnodeBody->sxBlock.scope;

ParseNode\* listNode = Parser::StaticCreateBinNode(knopList, nullptr, nullptr, alloc);

outerBod->sxBlock.pnodeStmt = listNode;

ParseNode\* innerBodOriginal = innerLoop->sxFor.pnodeBody;

ParseNode\* origStmt = innerBodOriginal->sxBlock.pnodeStmt;

if (origStmt->nop == knopList)

{

ParseNode\* invertedStmt = nullptr;

while (origStmt->nop == knopList)

{

ParseNode\* invertedItem = ConstructInvertedStatement(origStmt->sxBin.pnode1, byteCodeGenerator, funcInfo, &listNode);

if (invertedStmt != nullptr)

{

invertedStmt = invertedStmt->sxBin.pnode2 = byteCodeGenerator->GetParser()->CreateBinNode(knopList, invertedItem, nullptr);

}

else

{

invertedStmt = innerBod->sxBlock.pnodeStmt = byteCodeGenerator->GetParser()->CreateBinNode(knopList, invertedItem, nullptr);

}

origStmt = origStmt->sxBin.pnode2;

}

Assert(invertedStmt != nullptr);

invertedStmt->sxBin.pnode2 = ConstructInvertedStatement(origStmt, byteCodeGenerator, funcInfo, &listNode);

}

else

{

innerBod->sxBlock.pnodeStmt = ConstructInvertedStatement(origStmt, byteCodeGenerator, funcInfo, &listNode);

}

if (listNode->sxBin.pnode1 == nullptr)

{

listNode->sxBin.pnode1 = byteCodeGenerator->GetParser()->CreateTempNode(nullptr);

}

listNode->sxBin.pnode2 = innerLoopC;

return outerLoopC;

}

bool InvertableStmt(ParseNode\* stmt, Symbol\* outerVar, ParseNode\* innerLoop, ParseNode\* outerLoop, ByteCodeGenerator\* byteCodeGenerator, SymCheck\* symCheck)

{

if (stmt != nullptr)

{

ParseNode\* lhs = nullptr;

ParseNode\* rhs = nullptr;

if (stmt->nop == knopAsg)

{

lhs = stmt->sxBin.pnode1;

rhs = stmt->sxBin.pnode2;

}

else if (stmt->nop == knopVarDecl)

{

rhs = stmt->sxVar.pnodeInit;

}

if (lhs != nullptr)

{

if (lhs->nop == knopDot)

{

return false;

}

if (lhs->nop == knopName)

{

if ((lhs->sxPid.sym != nullptr) && (lhs->sxPid.sym->GetIsGlobal()))

{

return false;

}

}

else if (lhs->nop == knopIndex)

{

ParseNode\* indexed = lhs->sxBin.pnode1;

ParseNode\* index = lhs->sxBin.pnode2;

if ((index == nullptr) || (indexed == nullptr))

{

return false;

}

if ((indexed->nop != knopName) || (indexed->sxPid.sym == nullptr))

{

return false;

}

if (!InvertableExprPlus(symCheck, index, byteCodeGenerator, outerVar))

{

return false;

}

}

}

if (rhs != nullptr)

{

if (!InvertableExpr(symCheck, rhs, byteCodeGenerator))

{

return false;

}

}

else

{

if (!InvertableExpr(symCheck, stmt, byteCodeGenerator))

{

return false;

}

}

return true;

}

return false;

}

bool GatherInversionSyms(ParseNode\* stmt, Symbol\* outerVar, ParseNode\* innerLoop, ByteCodeGenerator\* byteCodeGenerator, SymCheck\* symCheck)

{

if (stmt != nullptr)

{

ParseNode\* lhs = nullptr;

Symbol\* auxSym = nullptr;

if (stmt->nop == knopAsg)

{

lhs = stmt->sxBin.pnode1;

}

else if (stmt->nop == knopVarDecl)

{

auxSym = stmt->sxVar.sym;

}

if (lhs != nullptr)

{

if (lhs->nop == knopDot)

{

return false;

}

if (lhs->nop == knopName)

{

if ((lhs->sxPid.sym == nullptr) || (lhs->sxPid.sym->GetIsGlobal()))

{

return false;

}

else

{

auxSym = lhs->sxPid.sym;

}

}

}

if (auxSym != nullptr)

{

return symCheck->AddSymbol(auxSym);

}

}

return true;

}

bool InvertableBlock(ParseNode\* block, Symbol\* outerVar, ParseNode\* innerLoop, ParseNode\* outerLoop, ByteCodeGenerator\* byteCodeGenerator,

SymCheck\* symCheck)

{

if (block == nullptr)

{

return false;

}

if (!symCheck->AddSymbol(outerVar))

{

return false;

}

if (innerLoop->sxFor.pnodeBody->nop == knopBlock && innerLoop->sxFor.pnodeBody->sxBlock.HasBlockScopedContent()

|| outerLoop->sxFor.pnodeBody->nop == knopBlock && outerLoop->sxFor.pnodeBody->sxBlock.HasBlockScopedContent())

{

// we can not invert loops if there are block scoped declarations inside

return false;

}

if ((block != nullptr) && (block->nop == knopBlock))

{

ParseNode\* stmt = block->sxBlock.pnodeStmt;

while ((stmt != nullptr) && (stmt->nop == knopList))

{

if (!GatherInversionSyms(stmt->sxBin.pnode1, outerVar, innerLoop, byteCodeGenerator, symCheck))

{

return false;

}

stmt = stmt->sxBin.pnode2;

}

if (!GatherInversionSyms(stmt, outerVar, innerLoop, byteCodeGenerator, symCheck))

{

return false;

}

stmt = block->sxBlock.pnodeStmt;

while ((stmt != nullptr) && (stmt->nop == knopList))

{

if (!InvertableStmt(stmt->sxBin.pnode1, outerVar, innerLoop, outerLoop, byteCodeGenerator, symCheck))

{

return false;

}

stmt = stmt->sxBin.pnode2;

}

if (!InvertableStmt(stmt, outerVar, innerLoop, outerLoop, byteCodeGenerator, symCheck))

{

return false;

}

return (InvertableExprPlus(symCheck, innerLoop->sxFor.pnodeCond, byteCodeGenerator, nullptr) &&

InvertableExprPlus(symCheck, outerLoop->sxFor.pnodeCond, byteCodeGenerator, outerVar));

}

else

{

return false;

}

}

// Start of invert loop optimization.

// For now, find simple cases (only for loops around single assignment).

// Returns new AST for inverted loop; also returns in out param

// side effects level, if any that guards the new AST (old AST will be

// used if guard fails).

// Should only be called with loopNode representing top-level statement.

ParseNode\* InvertLoop(ParseNode\* outerLoop, ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* funcInfo)

{

if (byteCodeGenerator->GetScriptContext()->optimizationOverrides.GetSideEffects() != Js::SideEffects\_None)

{

return nullptr;

}

SymCheck symCheck;

symCheck.Init();

if (outerLoop->nop == knopFor)

{

ParseNode\* innerLoop = outerLoop->sxFor.pnodeBody;

if ((innerLoop == nullptr) || (innerLoop->nop != knopBlock))

{

return nullptr;

}

else

{

innerLoop = innerLoop->sxBlock.pnodeStmt;

}

if ((innerLoop != nullptr) && (innerLoop->nop == knopFor))

{

if ((outerLoop->sxFor.pnodeInit != nullptr) &&

(outerLoop->sxFor.pnodeInit->nop == knopVarDecl) &&

(outerLoop->sxFor.pnodeInit->sxVar.pnodeInit != nullptr) &&

(outerLoop->sxFor.pnodeInit->sxVar.pnodeInit->nop == knopInt) &&

(outerLoop->sxFor.pnodeIncr != nullptr) &&

((outerLoop->sxFor.pnodeIncr->nop == knopIncPre) || (outerLoop->sxFor.pnodeIncr->nop == knopIncPost)) &&

(outerLoop->sxFor.pnodeIncr->sxUni.pnode1->nop == knopName) &&

(outerLoop->sxFor.pnodeInit->sxVar.pid == outerLoop->sxFor.pnodeIncr->sxUni.pnode1->sxPid.pid) &&

(innerLoop->sxFor.pnodeIncr != nullptr) &&

((innerLoop->sxFor.pnodeIncr->nop == knopIncPre) || (innerLoop->sxFor.pnodeIncr->nop == knopIncPost)) &&

(innerLoop->sxFor.pnodeInit != nullptr) &&

(innerLoop->sxFor.pnodeInit->nop == knopVarDecl) &&

(innerLoop->sxFor.pnodeInit->sxVar.pnodeInit != nullptr) &&

(innerLoop->sxFor.pnodeInit->sxVar.pnodeInit->nop == knopInt) &&

(innerLoop->sxFor.pnodeIncr->sxUni.pnode1->nop == knopName) &&

(innerLoop->sxFor.pnodeInit->sxVar.pid == innerLoop->sxFor.pnodeIncr->sxUni.pnode1->sxPid.pid))

{

Symbol\* outerVar = outerLoop->sxFor.pnodeInit->sxVar.sym;

Symbol\* innerVar = innerLoop->sxFor.pnodeInit->sxVar.sym;

if ((outerVar != nullptr) && (innerVar != nullptr))

{

ParseNode\* block = innerLoop->sxFor.pnodeBody;

if (InvertableBlock(block, outerVar, innerLoop, outerLoop, byteCodeGenerator, &symCheck))

{

return ConstructInvertedLoop(innerLoop, outerLoop, byteCodeGenerator, funcInfo);

}

}

}

}

}

return nullptr;

}

void SetAdditionalBindInfoForVariables(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator)

{

Symbol \*sym = pnode->sxVar.sym;

if (sym == nullptr)

{

return;

}

FuncInfo\* func = byteCodeGenerator->TopFuncInfo();

if (func->IsGlobalFunction())

{

func->SetHasGlobalRef(true);

}

if (!sym->GetIsGlobal() && !sym->GetIsArguments() &&

(sym->GetScope() == func->GetBodyScope() || sym->GetScope() == func->GetParamScope() || sym->GetScope()->GetCanMerge()))

{

if (func->GetChildCallsEval())

{

func->SetHasLocalInClosure(true);

}

else

{

sym->RecordDef();

}

}

// If this decl does an assignment inside a loop body, then there's a chance

// that a jitted loop body will expect us to begin with a valid value in this var.

// So mark the sym as used so that we guarantee the var will at least get "undefined".

if (byteCodeGenerator->IsInLoop() &&

pnode->sxVar.pnodeInit)

{

sym->SetIsUsed(true);

}

}

// bind references to definitions (prefix pass)

void Bind(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator)

{

if (pnode == nullptr)

{

return;

}

switch (pnode->nop)

{

case knopBreak:

case knopContinue:

byteCodeGenerator->AddTargetStmt(pnode->sxJump.pnodeTarget);

break;

case knopProg:

{

FuncInfo\* globFuncInfo = byteCodeGenerator->StartBindGlobalStatements(pnode);

pnode->sxFnc.funcInfo = globFuncInfo;

AddFunctionsToScope(pnode->sxFnc.GetTopLevelScope(), byteCodeGenerator);

AddVarsToScope(pnode->sxFnc.pnodeVars, byteCodeGenerator);

// There are no args to add, but "eval" gets a this pointer.

byteCodeGenerator->SetNumberOfInArgs(!!(byteCodeGenerator->GetFlags() & fscrEvalCode));

if (!globFuncInfo->IsFakeGlobalFunction(byteCodeGenerator->GetFlags()))

{

// Global code: the root function is the global function.

byteCodeGenerator->SetRootFuncInfo(globFuncInfo);

}

else if (globFuncInfo->byteCodeFunction)

{

// If the current global code wasn't marked to be treated as global code (e.g. from deferred parsing),

// we don't need to send a register script event for it.

globFuncInfo->byteCodeFunction->SetIsTopLevel(false);

}

if (pnode->sxFnc.CallsEval())

{

globFuncInfo->SetCallsEval(true);

}

break;

}

case knopFncDecl:

// VisitFunctionsInScope has already done binding within the declared function. Here, just record the fact

// that the parent function has a local/global declaration in it.

BindFuncSymbol(pnode, byteCodeGenerator);

if (pnode->sxFnc.IsGenerator())

{

// Always assume generator functions escape since tracking them requires tracking

// the resulting generators in addition to the function.

byteCodeGenerator->FuncEscapes(byteCodeGenerator->TopFuncInfo()->GetBodyScope());

}

if (!pnode->sxFnc.IsDeclaration())

{

FuncInfo \*funcInfo = byteCodeGenerator->TopFuncInfo();

if (!funcInfo->IsGlobalFunction() || (byteCodeGenerator->GetFlags() & fscrEval))

{

// In the case of a nested function expression, assumes that it escapes.

// We could try to analyze what it touches to be more precise.

byteCodeGenerator->FuncEscapes(funcInfo->GetBodyScope());

}

byteCodeGenerator->ProcessCapturedSyms(pnode);

}

else if (byteCodeGenerator->IsInLoop())

{

Symbol \*funcSym = pnode->sxFnc.GetFuncSymbol();

if (funcSym)

{

Symbol \*funcVarSym = funcSym->GetFuncScopeVarSym();

if (funcVarSym)

{

// We're going to write to the funcVarSym when we do the function instantiation,

// so treat the funcVarSym as used. That way, we know it will get undef-initialized at the

// top of the function, so a jitted loop body won't have any issue with boxing if

// the function instantiation isn't executed.

Assert(funcVarSym != funcSym);

funcVarSym->SetIsUsed(true);

}

}

}

break;

case knopThis:

case knopSuper:

{

FuncInfo \*top = byteCodeGenerator->TopFuncInfo();

if (top->IsGlobalFunction() && !(byteCodeGenerator->GetFlags() & fscrEval))

{

top->SetHasGlobalRef(true);

}

else if (top->IsLambda())

{

byteCodeGenerator->MarkThisUsedInLambda();

}

// "this" should be loaded for both global and non-global functions

byteCodeGenerator->TopFuncInfo()->GetParsedFunctionBody()->SetHasThis(true);

break;

}

case knopName:

{

if (pnode->sxPid.sym == nullptr)

{

if (pnode->grfpn & fpnMemberReference)

{

// This is a member name. No binding.

break;

}

Symbol \*sym = byteCodeGenerator->FindSymbol(pnode->sxPid.symRef, pnode->sxPid.pid);

if (sym)

{

// This is a named load, not just a reference, so if it's a nested function note that all

// the nested scopes escape.

Assert(!sym->GetDecl() || (pnode->sxPid.symRef && \*pnode->sxPid.symRef));

Assert(!sym->GetDecl() || ((\*pnode->sxPid.symRef)->GetDecl() == sym->GetDecl()));

pnode->sxPid.sym = sym;

if (sym->GetSymbolType() == STFunction &&

(!sym->GetIsGlobal() || (byteCodeGenerator->GetFlags() & fscrEval)))

{

byteCodeGenerator->FuncEscapes(sym->GetScope());

}

}

}

FuncInfo \*top = byteCodeGenerator->TopFuncInfo();

if (pnode->sxPid.sym == nullptr || pnode->sxPid.sym->GetIsGlobal())

{

top->SetHasGlobalRef(true);

}

if (pnode->sxPid.sym)

{

pnode->sxPid.sym->SetIsUsed(true);

}

break;

}

case knopMember:

case knopMemberShort:

case knopObjectPatternMember:

if (pnode->sxBin.pnode1->nop == knopComputedName)

{

// Computed property name - cannot bind yet

break;

}

// fall through

case knopGetMember:

case knopSetMember:

{

// lhs is knopStr, rhs is expr

ParseNode \*id = pnode->sxBin.pnode1;

if (id->nop == knopStr || id->nop == knopName)

{

byteCodeGenerator->AssignPropertyId(id->sxPid.pid);

id->sxPid.sym = nullptr;

id->sxPid.symRef = nullptr;

id->grfpn |= fpnMemberReference;

}

break;

}

// TODO: convert index over string to Get/Put Value

case knopIndex:

BindReference(pnode, byteCodeGenerator);

break;

case knopDot:

BindInstAndMember(pnode, byteCodeGenerator);

break;

case knopTryFinally:

byteCodeGenerator->SetHasFinally(true);

case knopTryCatch:

byteCodeGenerator->SetHasTry(true);

byteCodeGenerator->TopFuncInfo()->byteCodeFunction->SetDontInline(true);

byteCodeGenerator->AddTargetStmt(pnode);

break;

case knopAsg:

BindReference(pnode, byteCodeGenerator);

CheckLocalVarDef(pnode, byteCodeGenerator);

break;

case knopVarDecl:

// "arguments" symbol or decl w/o RHS may have been bound already; otherwise, do the binding here.

if (pnode->sxVar.sym == nullptr)

{

pnode->sxVar.sym = byteCodeGenerator->FindSymbol(pnode->sxVar.symRef, pnode->sxVar.pid);

}

SetAdditionalBindInfoForVariables(pnode, byteCodeGenerator);

break;

case knopConstDecl:

case knopLetDecl:

// "arguments" symbol or decl w/o RHS may have been bound already; otherwise, do the binding here.

if (!pnode->sxVar.sym)

{

AssertMsg(pnode->sxVar.symRef && \*pnode->sxVar.symRef, "'const' and 'let' should be binded when we bind block");

pnode->sxVar.sym = \*pnode->sxVar.symRef;

}

SetAdditionalBindInfoForVariables(pnode, byteCodeGenerator);

break;

case knopCall:

if (pnode->sxCall.isEvalCall && byteCodeGenerator->TopFuncInfo()->IsLambda())

{

byteCodeGenerator->MarkThisUsedInLambda();

}

// fallthrough

case knopTypeof:

case knopDelete:

BindReference(pnode, byteCodeGenerator);

break;

case knopRegExp:

pnode->sxPid.regexPatternIndex = byteCodeGenerator->TopFuncInfo()->GetParsedFunctionBody()->NewLiteralRegex();

break;

case knopComma:

pnode->sxBin.pnode1->SetNotEscapedUse();

break;

case knopBlock:

{

for (ParseNode \*pnodeScope = pnode->sxBlock.pnodeScopes; pnodeScope; /\* no increment \*/)

{

switch (pnodeScope->nop)

{

case knopFncDecl:

if (pnodeScope->sxFnc.IsDeclaration())

{

byteCodeGenerator->ProcessCapturedSyms(pnodeScope);

}

pnodeScope = pnodeScope->sxFnc.pnodeNext;

break;

case knopBlock:

pnodeScope = pnodeScope->sxBlock.pnodeNext;

break;

case knopCatch:

pnodeScope = pnodeScope->sxCatch.pnodeNext;

break;

case knopWith:

pnodeScope = pnodeScope->sxWith.pnodeNext;

break;

}

}

break;

}

}

}

void ByteCodeGenerator::ProcessCapturedSyms(ParseNode \*pnode)

{

SymbolTable \*capturedSyms = pnode->sxFnc.funcInfo->GetCapturedSyms();

if (capturedSyms)

{

FuncInfo \*funcInfo = this->TopFuncInfo();

CapturedSymMap \*capturedSymMap = funcInfo->EnsureCapturedSymMap();

ParseNode \*pnodeStmt = this->GetCurrentTopStatement();

SList<Symbol\*> \*capturedSymList;

if (!pnodeStmt->CapturesSyms())

{

capturedSymList = Anew(this->alloc, SList<Symbol\*>, this->alloc);

capturedSymMap->Add(pnodeStmt, capturedSymList);

pnodeStmt->SetCapturesSyms();

}

else

{

capturedSymList = capturedSymMap->Item(pnodeStmt);

}

capturedSyms->Map([&](Symbol \*sym)

{

if (!sym->GetIsCommittedToSlot() && !sym->HasVisitedCapturingFunc())

{

capturedSymList->Prepend(sym);

sym->SetHasVisitedCapturingFunc();

}

});

}

}

void ByteCodeGenerator::MarkThisUsedInLambda()

{

// This is a lambda that refers to "this".

// Find the enclosing "normal" function and indicate that the lambda captures the enclosing function's "this".

FuncInfo \*parent = this->FindEnclosingNonLambda();

parent->GetParsedFunctionBody()->SetHasThis(true);

if (!parent->IsGlobalFunction() || this->GetFlags() & fscrEval)

{

// If the enclosing function is non-global or eval global, it will put "this" in a closure slot.

parent->SetIsThisLexicallyCaptured();

Scope\* scope = parent->IsGlobalFunction() ? parent->GetGlobalEvalBlockScope() : parent->GetBodyScope();

scope->SetHasLocalInClosure(true);

this->TopFuncInfo()->SetHasClosureReference(true);

}

this->TopFuncInfo()->SetHasCapturedThis();

}

void ByteCodeGenerator::FuncEscapes(Scope \*scope)

{

while (scope)

{

Assert(scope->GetFunc());

scope->GetFunc()->SetEscapes(true);

scope = scope->GetEnclosingScope();

}

if (this->flags & fscrEval)

{

// If a function declared inside eval escapes, we'll need

// to invalidate the caller's cached scope.

this->funcEscapes = true;

}

}

bool ByteCodeGenerator::HasInterleavingDynamicScope(Symbol \* sym) const

{

Js::PropertyId unused;

return this->InDynamicScope() &&

sym->GetScope() != this->FindScopeForSym(sym->GetScope(), nullptr, &unused, this->TopFuncInfo());

}

void CheckMaybeEscapedUse(ParseNode \* pnode, ByteCodeGenerator \* byteCodeGenerator, bool isCall = false)

{

if (pnode == nullptr)

{

return;

}

FuncInfo \* topFunc = byteCodeGenerator->TopFuncInfo();

if (topFunc->IsGlobalFunction())

{

return;

}

switch (pnode->nop)

{

case knopAsg:

if (pnode->sxBin.pnode1->nop != knopName)

{

break;

}

// use of an assignment (e.g. (y = function() {}) + "1"), just make y an escaped use.

pnode = pnode->sxBin.pnode1;

isCall = false;

// fall-through

case knopName:

if (!isCall)

{

// Mark the name has having escaped use

if (pnode->sxPid.sym)

{

pnode->sxPid.sym->SetHasMaybeEscapedUse(byteCodeGenerator);

}

}

break;

case knopFncDecl:

// A function declaration has an unknown use (not assignment nor call),

// mark the function as having child escaped

topFunc->SetHasMaybeEscapedNestedFunc(DebugOnly(L"UnknownUse"));

break;

}

}

void CheckFuncAssignment(Symbol \* sym, ParseNode \* pnode2, ByteCodeGenerator \* byteCodeGenerator)

{

if (pnode2 == nullptr)

{

return;

}

switch (pnode2->nop)

{

default:

CheckMaybeEscapedUse(pnode2, byteCodeGenerator);

break;

case knopFncDecl:

{

FuncInfo \* topFunc = byteCodeGenerator->TopFuncInfo();

if (topFunc->IsGlobalFunction())

{

return;

}

// Use not as an assignment or assignment to an outer function's sym, or assigned to a formal

// or assigned to multiple names.

if (sym == nullptr

|| sym->GetScope()->GetFunc() != topFunc)

{

topFunc->SetHasMaybeEscapedNestedFunc(DebugOnly(

sym == nullptr ? L"UnknownAssignment" :

(sym->GetScope()->GetFunc() != topFunc) ? L"CrossFuncAssignment" :

L"SomethingIsWrong!")

);

}

else

{

// TODO-STACK-NESTED-FUNC: Since we only support single def functions, we can still put the

// nested function on the stack and reuse even if the function goes out of the block scope.

// However, we cannot allocate frame display or slots on the stack if the function is

// declared in a loop, because there might be multiple functions referencing different

// iterations of the scope.

// For now, just disable everything.

Scope \* funcParentScope = pnode2->sxFnc.funcInfo->GetBodyScope()->GetEnclosingScope();

while (sym->GetScope() != funcParentScope)

{

if (funcParentScope->GetMustInstantiate())

{

topFunc->SetHasMaybeEscapedNestedFunc(DebugOnly(L"CrossScopeAssignment"));

break;

}

funcParentScope->SetHasCrossScopeFuncAssignment();

funcParentScope = funcParentScope->GetEnclosingScope();

}

// Need to always detect interleaving dynamic scope ('with') for assignments

// as those may end up escaping into the 'with' scope.

// TODO: the with scope is marked as MustInstaniate late during byte code emit

// We could detect this using the loop above as well, by marking the with

// scope as must instantiate early, this is just less risky of a fix for RTM.

if (byteCodeGenerator->HasInterleavingDynamicScope(sym))

{

byteCodeGenerator->TopFuncInfo()->SetHasMaybeEscapedNestedFunc(DebugOnly(L"InterleavingDynamicScope"));

}

sym->SetHasFuncAssignment(byteCodeGenerator);

}

}

break;

};

}

inline bool ContainsSuperReference(ParseNodePtr pnode)

{

return (pnode->sxCall.pnodeTarget->nop == knopDot && pnode->sxCall.pnodeTarget->sxBin.pnode1->nop == knopSuper) // super.prop()

|| (pnode->sxCall.pnodeTarget->nop == knopIndex && pnode->sxCall.pnodeTarget->sxBin.pnode1->nop == knopSuper); // super[prop]()

}

inline bool ContainsDirectSuper(ParseNodePtr pnode)

{

return pnode->sxCall.pnodeTarget->nop == knopSuper; // super()

}

// Assign permanent (non-temp) registers for the function.

// These include constants (null, 3.7, this) and locals that use registers as their home locations.

// Assign the location fields of parse nodes whose values are constants/locals with permanent/known registers.

// Re-usable expression temps are assigned during the final Emit pass.

void AssignRegisters(ParseNode \*pnode, ByteCodeGenerator \*byteCodeGenerator)

{

if (pnode == nullptr)

{

return;

}

Symbol \*sym;

OpCode nop = pnode->nop;

switch (nop)

{

default:

{

uint flags = ParseNode::Grfnop(nop);

if (flags & fnopUni)

{

CheckMaybeEscapedUse(pnode->sxUni.pnode1, byteCodeGenerator);

}

else if (flags & fnopBin)

{

CheckMaybeEscapedUse(pnode->sxBin.pnode1, byteCodeGenerator);

CheckMaybeEscapedUse(pnode->sxBin.pnode2, byteCodeGenerator);

}

break;

}

case knopParamPattern:

byteCodeGenerator->AssignUndefinedConstRegister();

CheckMaybeEscapedUse(pnode->sxParamPattern.pnode1, byteCodeGenerator);

break;

case knopArrayPattern:

byteCodeGenerator->AssignUndefinedConstRegister();

CheckMaybeEscapedUse(pnode->sxUni.pnode1, byteCodeGenerator);

break;

case knopDot:

CheckMaybeEscapedUse(pnode->sxBin.pnode1, byteCodeGenerator);

break;

case knopMember:

case knopMemberShort:

case knopGetMember:

case knopSetMember:

CheckMaybeEscapedUse(pnode->sxBin.pnode2, byteCodeGenerator);

break;

case knopAsg:

{

Symbol \* sym = pnode->sxBin.pnode1->nop == knopName ? pnode->sxBin.pnode1->sxPid.sym : nullptr;

CheckFuncAssignment(sym, pnode->sxBin.pnode2, byteCodeGenerator);

if (pnode->IsInList())

{

// Assignment in array literal

CheckMaybeEscapedUse(pnode->sxBin.pnode1, byteCodeGenerator);

}

if (byteCodeGenerator->IsES6DestructuringEnabled() && (pnode->sxBin.pnode1->nop == knopArrayPattern || pnode->sxBin.pnode1->nop == knopObjectPattern))

{

// Destructured arrays may have default values and need undefined.

byteCodeGenerator->AssignUndefinedConstRegister();

// Any rest parameter in a destructured array will need a 0 constant.

byteCodeGenerator->EnregisterConstant(0);

}

break;

}

case knopEllipsis:

if (byteCodeGenerator->InDestructuredPattern())

{

// Get a register for the rest array counter.

pnode->location = byteCodeGenerator->NextVarRegister();

// Any rest parameter in a destructured array will need a 0 constant.

byteCodeGenerator->EnregisterConstant(0);

}

CheckMaybeEscapedUse(pnode->sxUni.pnode1, byteCodeGenerator);

break;

case knopQmark:

CheckMaybeEscapedUse(pnode->sxTri.pnode1, byteCodeGenerator);

CheckMaybeEscapedUse(pnode->sxTri.pnode2, byteCodeGenerator);

CheckMaybeEscapedUse(pnode->sxTri.pnode3, byteCodeGenerator);

break;

case knopWith:

pnode->location = byteCodeGenerator->NextVarRegister();

CheckMaybeEscapedUse(pnode->sxWith.pnodeObj, byteCodeGenerator);

break;

case knopComma:

if (!pnode->IsNotEscapedUse())

{

// Only the last expr in comma expr escape. Mark it if it is escapable.

CheckMaybeEscapedUse(pnode->sxBin.pnode2, byteCodeGenerator);

}

break;

case knopFncDecl:

if (!byteCodeGenerator->TopFuncInfo()->IsGlobalFunction())

{

if (pnode->sxFnc.IsGenerator())

{

// Assume generators always escape; otherwise need to analyze if

// the return value of calls to generator function, the generator

// objects, escape.

FuncInfo\* funcInfo = byteCodeGenerator->TopFuncInfo();

funcInfo->SetHasMaybeEscapedNestedFunc(DebugOnly(L"Generator"));

}

if (pnode->IsInList() && !pnode->IsNotEscapedUse())

{

byteCodeGenerator->TopFuncInfo()->SetHasMaybeEscapedNestedFunc(DebugOnly(L"InList"));

}

ParseNodePtr pnodeName = pnode->sxFnc.pnodeName;

if (pnodeName != nullptr)

{

// REVIEW: does this apply now that compat mode is gone?

// There is a weird case in compat mode where we may not have a sym assigned to a fnc decl's

// name node if it is a named function declare inside 'with' that also assigned to something else

// as well. Instead, We generate two knopFncDecl node one for parent function and one for the assignment.

// Only the top one gets a sym, not the inner one. The assignment in the 'with' will be using the inner

// one. Also we will detect that the assignment to a variable is an escape inside a 'with'.

// Since we need the sym in the fnc decl's name, we just detect the escape here as "WithScopeFuncName".

if (pnodeName->nop == knopVarDecl && pnodeName->sxVar.sym != nullptr)

{

// Unlike in CheckFuncAssignemnt, we don't have check if there is a interleaving

// dynamic scope ('with') here, because we also generate direct assignment for

// function decl's names

pnodeName->sxVar.sym->SetHasFuncAssignment(byteCodeGenerator);

// Function declaration in block scope and non-strict mode has a

// corresponding var sym that we assign to as well. Need to

// mark that symbol as has func assignment as well.

Symbol \* functionScopeVarSym = pnodeName->sxVar.sym->GetFuncScopeVarSym();

if (functionScopeVarSym)

{

functionScopeVarSym->SetHasFuncAssignment(byteCodeGenerator);

}

}

else

{

// The function has multiple names, or assign to o.x or o::x

byteCodeGenerator->TopFuncInfo()->SetHasMaybeEscapedNestedFunc(DebugOnly(

pnodeName->nop == knopList ? L"MultipleFuncName" :

pnodeName->nop == knopDot ? L"PropFuncName" :

pnodeName->nop == knopVarDecl && pnodeName->sxVar.sym == nullptr ? L"WithScopeFuncName" :

L"WeirdFuncName"

));

}

}

}

break;

case knopNew:

CheckMaybeEscapedUse(pnode->sxCall.pnodeTarget, byteCodeGenerator);

CheckMaybeEscapedUse(pnode->sxCall.pnodeArgs, byteCodeGenerator);

break;

case knopThrow:

CheckMaybeEscapedUse(pnode->sxUni.pnode1, byteCodeGenerator);

break;

// REVIEW: Technically, switch expr or case expr doesn't really escape as strict equal

// doesn't cause the function to escape.

case knopSwitch:

CheckMaybeEscapedUse(pnode->sxSwitch.pnodeVal, byteCodeGenerator);

break;

case knopCase:

CheckMaybeEscapedUse(pnode->sxCase.pnodeExpr, byteCodeGenerator);

break;

// REVIEW: Technically, the object for GetForInEnumerator doesn't escape, except when cached,

// which we can make work.

case knopForIn:

CheckMaybeEscapedUse(pnode->sxForInOrForOf.pnodeObj, byteCodeGenerator);

break;

case knopForOf:

byteCodeGenerator->AssignNullConstRegister();

byteCodeGenerator->AssignUndefinedConstRegister();

CheckMaybeEscapedUse(pnode->sxForInOrForOf.pnodeObj, byteCodeGenerator);

break;

case knopTrue:

pnode->location = byteCodeGenerator->AssignTrueConstRegister();

break;

case knopFalse:

pnode->location = byteCodeGenerator->AssignFalseConstRegister();

break;

case knopDecPost:

case knopIncPost:

case knopDecPre:

case knopIncPre:

byteCodeGenerator->EnregisterConstant(1);

CheckMaybeEscapedUse(pnode->sxUni.pnode1, byteCodeGenerator);

break;

case knopObject:

byteCodeGenerator->AssignNullConstRegister();

break;

case knopClassDecl:

{

FuncInfo \* topFunc = byteCodeGenerator->TopFuncInfo();

topFunc->SetHasMaybeEscapedNestedFunc(DebugOnly(L"Class"));

// We may need undefined for the 'this', e.g. calling a class expression

byteCodeGenerator->AssignUndefinedConstRegister();

break;

}

case knopNull:

pnode->location = byteCodeGenerator->AssignNullConstRegister();

break;

case knopThis:

{

FuncInfo\* func = byteCodeGenerator->TopFuncInfo();

pnode->location = func->AssignThisRegister();

if (func->IsLambda())

{

func = byteCodeGenerator->FindEnclosingNonLambda();

func->AssignThisRegister();

if (func->IsGlobalFunction() && !(byteCodeGenerator->GetFlags() & fscrEval))

{

byteCodeGenerator->AssignNullConstRegister();

}

}

// "this" should be loaded for both global and non global functions

if (func->IsGlobalFunction() && !(byteCodeGenerator->GetFlags() & fscrEval))

{

// We'll pass "null" to LdThis, to simulate "null" passed as "this" to the

// global function.

func->AssignNullConstRegister();

}

break;

}

case knopNewTarget:

{

FuncInfo\* func = byteCodeGenerator->TopFuncInfo();

pnode->location = func->AssignNewTargetRegister();

FuncInfo\* nonLambdaFunc = func;

if (func->IsLambda())

{

nonLambdaFunc = byteCodeGenerator->FindEnclosingNonLambda();

}

if (nonLambdaFunc != func || (func->IsGlobalFunction() && (byteCodeGenerator->GetFlags() & fscrEval)))

{

nonLambdaFunc->root->sxFnc.SetHasNewTargetReferene();

nonLambdaFunc->AssignNewTargetRegister();

nonLambdaFunc->SetIsNewTargetLexicallyCaptured();

nonLambdaFunc->GetBodyScope()->SetHasLocalInClosure(true);

func->SetHasClosureReference(true);

}

break;

}

case knopSuper:

{

FuncInfo\* func = byteCodeGenerator->TopFuncInfo();

pnode->location = func->AssignSuperRegister();

func->AssignThisRegister();

FuncInfo\* nonLambdaFunc = func;

if (func->IsLambda())

{

// If this is a lambda inside a class member, the class member will need to load super.

nonLambdaFunc = byteCodeGenerator->FindEnclosingNonLambda();

nonLambdaFunc->root->sxFnc.SetHasSuperReference();

nonLambdaFunc->AssignSuperRegister();

nonLambdaFunc->AssignThisRegister();

nonLambdaFunc->SetIsSuperLexicallyCaptured();

if (nonLambdaFunc->IsClassConstructor())

{

func->AssignNewTargetRegister();

nonLambdaFunc->root->sxFnc.SetHasNewTargetReferene();

nonLambdaFunc->AssignNewTargetRegister();

nonLambdaFunc->SetIsNewTargetLexicallyCaptured();

nonLambdaFunc->AssignUndefinedConstRegister();

}

nonLambdaFunc->GetBodyScope()->SetHasLocalInClosure(true);

func->SetHasClosureReference(true);

}

else

{

if (func->IsClassConstructor())

{

func->AssignNewTargetRegister();

}

}

if (nonLambdaFunc->IsGlobalFunction())

{

if (!(byteCodeGenerator->GetFlags() & fscrEval))

{

// Enable LdSuper for global function to support subsequent emission of call, dot, prop, etc., related to super.

func->AssignNullConstRegister();

nonLambdaFunc->AssignNullConstRegister();

}

}

else if (!func->IsClassMember())

{

func->AssignUndefinedConstRegister();

}

break;

}

case knopCall:

{

if (pnode->sxCall.pnodeTarget->nop != knopIndex &&

pnode->sxCall.pnodeTarget->nop != knopDot)

{

byteCodeGenerator->AssignUndefinedConstRegister();

}

bool containsDirectSuper = ContainsDirectSuper(pnode);

bool containsSuperReference = ContainsSuperReference(pnode);

if (containsDirectSuper)

{

pnode->sxCall.pnodeTarget->location = byteCodeGenerator->TopFuncInfo()->AssignSuperCtorRegister();

}

FuncInfo \*funcInfo = byteCodeGenerator->TopFuncInfo();

if (containsDirectSuper || containsSuperReference)

{

// A super call requires 'this' to be available.

byteCodeGenerator->SetNeedEnvRegister();

byteCodeGenerator->AssignThisRegister();

FuncInfo\* parent = funcInfo;

if (funcInfo->IsLambda())

{

// If this is a lambda inside a class member, the class member will need to load super.

parent = byteCodeGenerator->FindEnclosingNonLambda();

if (parent->root->sxFnc.IsClassMember())

{

// Set up super reference

if (containsSuperReference)

{

parent->root->sxFnc.SetHasSuperReference();

parent->AssignSuperRegister();

parent->SetIsSuperLexicallyCaptured();

}

else if (containsDirectSuper)

{

parent->root->sxFnc.SetHasDirectSuper();

parent->AssignSuperCtorRegister();

parent->SetIsSuperCtorLexicallyCaptured();

}

parent->GetBodyScope()->SetHasLocalInClosure(true);

funcInfo->SetHasClosureReference(true);

}

parent->AssignThisRegister();

byteCodeGenerator->MarkThisUsedInLambda();

}

// If this is a super call we need to have new.target

if (pnode->sxCall.pnodeTarget->nop == knopSuper)

{

byteCodeGenerator->AssignNewTargetRegister();

}

}

if (pnode->sxCall.isEvalCall)

{

if (!funcInfo->GetParsedFunctionBody()->IsReparsed())

{

Assert(funcInfo->IsGlobalFunction() || funcInfo->GetCallsEval());

funcInfo->SetCallsEval(true);

funcInfo->GetParsedFunctionBody()->SetCallsEval(true);

}

else

{

// On reparsing, load the state from function Body, instead of using the state on the parse node,

// as they might be different.

pnode->sxCall.isEvalCall = funcInfo->GetParsedFunctionBody()->GetCallsEval();

}

if (funcInfo->IsLambda() && pnode->sxCall.isEvalCall)

{

FuncInfo\* nonLambdaParent = byteCodeGenerator->FindEnclosingNonLambda();

if (!nonLambdaParent->IsGlobalFunction() || (byteCodeGenerator->GetFlags() & fscrEval))

{

nonLambdaParent->AssignThisRegister();

}

}

// An eval call in a class member needs to load super.

if (funcInfo->root->sxFnc.IsClassMember())

{

funcInfo->AssignSuperRegister();

if (funcInfo->root->sxFnc.IsClassConstructor() && !funcInfo->root->sxFnc.IsBaseClassConstructor())

{

funcInfo->AssignSuperCtorRegister();

}

}

else if (funcInfo->IsLambda())

{

// If this is a lambda inside a class member, the class member will need to load super.

FuncInfo \*parent = byteCodeGenerator->FindEnclosingNonLambda();

if (parent->root->sxFnc.IsClassMember())

{

parent->root->sxFnc.SetHasSuperReference();

parent->AssignSuperRegister();

if (parent->IsClassConstructor() && !parent->IsBaseClassConstructor())

{

parent->AssignSuperCtorRegister();

}

}

}

}

// Don't need to check pnode->sxCall.pnodeTarget even if it is a knopFncDecl,

// e.g. (function(){})();

// It is only used as a call, so don't count as an escape.

// Although not assigned to a slot, we will still able to box it by boxing

// all the stack function on the interpreter frame or the stack function link list

// on a jitted frame

break;

}

case knopInt:

pnode->location = byteCodeGenerator->EnregisterConstant(pnode->sxInt.lw);

break;

case knopFlt:

{

pnode->location = byteCodeGenerator->EnregisterDoubleConstant(pnode->sxFlt.dbl);

break;

}

case knopStr:

pnode->location = byteCodeGenerator->EnregisterStringConstant(pnode->sxPid.pid);

break;

case knopVarDecl:

case knopConstDecl:

case knopLetDecl:

{

sym = pnode->sxVar.sym;

Assert(sym != nullptr);

Assert(sym->GetScope()->GetEnclosingFunc() == byteCodeGenerator->TopFuncInfo());

if (pnode->sxVar.isBlockScopeFncDeclVar && sym->GetIsBlockVar())

{

break;

}

if (!sym->GetIsGlobal())

{

FuncInfo \*funcInfo = byteCodeGenerator->TopFuncInfo();

// Check the function assignment for the sym that we have, even if we remap it to function level sym below

// as we are going assign to the original sym

CheckFuncAssignment(sym, pnode->sxVar.pnodeInit, byteCodeGenerator);

if (sym->GetIsCatch() || (pnode->nop == knopVarDecl && sym->GetIsBlockVar() && !pnode->sxVar.isBlockScopeFncDeclVar))

{

// The LHS of the var decl really binds to the local symbol, not the catch or let symbol.

// But the assignment will go to the catch or let symbol. Just assign a register to the local

// so that it can get initialized to undefined.

#if DBG

if (!sym->GetIsCatch())

{

// Catch cannot be at function scope and let and var at function scope is redeclaration error.

Assert(funcInfo->bodyScope != sym->GetScope() || !byteCodeGenerator->GetScriptContext()->GetConfig()->IsBlockScopeEnabled());

}

#endif

auto symName = sym->GetName();

sym = funcInfo->bodyScope->FindLocalSymbol(symName);

if (sym == nullptr)

{

sym = funcInfo->paramScope->FindLocalSymbol(symName);

}

Assert((sym && !sym->GetIsCatch() && !sym->GetIsBlockVar()));

}

// Don't give the declared var a register if it's in a closure, because the closure slot

// is its true "home". (Need to check IsGlobal again as the sym may have changed above.)

if (!sym->GetIsGlobal() && !sym->IsInSlot(funcInfo))

{

if (PHASE\_TRACE(Js::DelayCapturePhase, funcInfo->byteCodeFunction))

{

if (sym->NeedsSlotAlloc(byteCodeGenerator->TopFuncInfo()))

{

Output::Print(L"--- DelayCapture: Delayed capturing symbol '%s' during initialization.\n", sym->GetName());

Output::Flush();

}

}

byteCodeGenerator->AssignRegister(sym);

}

}

else

{

Assert(byteCodeGenerator->TopFuncInfo()->IsGlobalFunction());

}

break;

}

case knopFor:

if ((pnode->sxFor.pnodeBody != nullptr) && (pnode->sxFor.pnodeBody->nop == knopBlock) &&

(pnode->sxFor.pnodeBody->sxBlock.pnodeStmt != nullptr) &&

(pnode->sxFor.pnodeBody->sxBlock.pnodeStmt->nop == knopFor) &&

(!byteCodeGenerator->IsInDebugMode()))

{

FuncInfo \*funcInfo = byteCodeGenerator->TopFuncInfo();

pnode->sxFor.pnodeInverted = InvertLoop(pnode, byteCodeGenerator, funcInfo);

}

else

{

pnode->sxFor.pnodeInverted = nullptr;

}

break;

case knopName:

sym = pnode->sxPid.sym;

if (sym == nullptr)

{

Assert(pnode->sxPid.pid->GetPropertyId() != Js::Constants::NoProperty);

}

else

{

// Note: don't give a register to a local if it's in a closure, because then the closure

// is its true home.

if (!sym->GetIsGlobal() &&

!sym->GetIsMember() &&

byteCodeGenerator->TopFuncInfo() == sym->GetScope()->GetEnclosingFunc() &&

!sym->IsInSlot(byteCodeGenerator->TopFuncInfo()) &&

!sym->HasVisitedCapturingFunc())

{

if (PHASE\_TRACE(Js::DelayCapturePhase, byteCodeGenerator->TopFuncInfo()->byteCodeFunction))

{

if (sym->NeedsSlotAlloc(byteCodeGenerator->TopFuncInfo()))

{

Output::Print(L"--- DelayCapture: Delayed capturing symbol '%s'.\n", sym->GetName());

Output::Flush();

}

}

// Local symbol being accessed in its own frame. Even if "with" or event

// handler semantics make the binding ambiguous, it has a home location,

// so assign it.

byteCodeGenerator->AssignRegister(sym);

// If we're in something like a "with" we'll need a scratch register to hold

// the multiple possible values of the property.

if (!byteCodeGenerator->HasInterleavingDynamicScope(sym))

{

// We're not in a dynamic scope, or our home scope is nested within the dynamic scope, so we

// don't have to do dynamic binding. Just use the home location for this reference.

pnode->location = sym->GetLocation();

}

}

}

if (pnode->IsInList() && !pnode->IsNotEscapedUse())

{

// A node that is in a list is assumed to be escape, unless marked otherwise.

// This includes array literal list/object literal list

CheckMaybeEscapedUse(pnode, byteCodeGenerator);

}

break;

case knopProg:

if (!byteCodeGenerator->HasParentScopeInfo())

{

// If we're compiling a nested deferred function, don't pop the scope stack,

// because we just want to leave it as-is for the emit pass.

PostVisitFunction(pnode, byteCodeGenerator);

}

break;

case knopReturn:

{

ParseNode \*pnodeExpr = pnode->sxReturn.pnodeExpr;

CheckMaybeEscapedUse(pnodeExpr, byteCodeGenerator);

break;

}

case knopStrTemplate:

{

ParseNode\* pnodeExprs = pnode->sxStrTemplate.pnodeSubstitutionExpressions;

if (pnodeExprs != nullptr)

{

while (pnodeExprs->nop == knopList)

{

Assert(pnodeExprs->sxBin.pnode1 != nullptr);

Assert(pnodeExprs->sxBin.pnode2 != nullptr);

CheckMaybeEscapedUse(pnodeExprs->sxBin.pnode1, byteCodeGenerator);

pnodeExprs = pnodeExprs->sxBin.pnode2;

}

// Also check the final element in the list

CheckMaybeEscapedUse(pnodeExprs, byteCodeGenerator);

}

if (pnode->sxStrTemplate.isTaggedTemplate)

{

pnode->location = byteCodeGenerator->EnregisterStringTemplateCallsiteConstant(pnode);

}

break;

}

case knopYieldLeaf:

byteCodeGenerator->AssignUndefinedConstRegister();

break;

case knopYield:

CheckMaybeEscapedUse(pnode->sxUni.pnode1, byteCodeGenerator);

break;

case knopYieldStar:

byteCodeGenerator->AssignNullConstRegister();

byteCodeGenerator->AssignUndefinedConstRegister();

CheckMaybeEscapedUse(pnode->sxUni.pnode1, byteCodeGenerator);

break;

}

}

// TODO[ianhall]: ApplyEnclosesArgs should be in ByteCodeEmitter.cpp but that becomes complicated because it depends on VisitIndirect

void PostCheckApplyEnclosesArgs(ParseNode\* pnode, ByteCodeGenerator\* byteCodeGenerator, ApplyCheck\* applyCheck);

void CheckApplyEnclosesArgs(ParseNode\* pnode, ByteCodeGenerator\* byteCodeGenerator, ApplyCheck\* applyCheck);

bool ApplyEnclosesArgs(ParseNode\* fncDecl, ByteCodeGenerator\* byteCodeGenerator)

{

if (byteCodeGenerator->IsInDebugMode())

{

// Inspection of the arguments object will be messed up if we do ApplyArgs.

return false;

}

if (!fncDecl->HasVarArguments()

&& fncDecl->sxFnc.pnodeArgs == nullptr

&& fncDecl->sxFnc.pnodeRest == nullptr

&& fncDecl->sxFnc.nestedCount == 0)

{

ApplyCheck applyCheck;

applyCheck.matches = true;

applyCheck.sawApply = false;

applyCheck.insideApplyCall = false;

VisitIndirect<ApplyCheck>(fncDecl->sxFnc.pnodeBody, byteCodeGenerator, &applyCheck, &CheckApplyEnclosesArgs, &PostCheckApplyEnclosesArgs);

return applyCheck.matches&&applyCheck.sawApply;

}

return false;

}

// TODO[ianhall]: VisitClearTmpRegs should be in ByteCodeEmitter.cpp but that becomes complicated because it depends on VisitIndirect

void ClearTmpRegs(ParseNode\* pnode, ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* emitFunc);

void VisitClearTmpRegs(ParseNode \* pnode, ByteCodeGenerator \* byteCodeGenerator, FuncInfo \* funcInfo)

{

VisitIndirect<FuncInfo>(pnode, byteCodeGenerator, funcInfo, &ClearTmpRegs, nullptr);

}

Js::FunctionBody \* ByteCodeGenerator::MakeGlobalFunctionBody(ParseNode \*pnode)

{

Js::FunctionBody \* func;

ENTER\_PINNED\_SCOPE(Js::PropertyRecordList, propertyRecordList);

propertyRecordList = EnsurePropertyRecordList();

func =

Js::FunctionBody::NewFromRecycler(

scriptContext,

Js::Constants::GlobalFunction,

Js::Constants::GlobalFunctionLength,

0,

pnode->sxFnc.nestedCount,

m\_utf8SourceInfo,

m\_utf8SourceInfo->GetSrcInfo()->sourceContextInfo->sourceContextId,

pnode->sxFnc.functionId,

propertyRecordList,

Js::FunctionInfo::Attributes::None

#ifdef PERF\_COUNTERS

, false /\* is function from deferred deserialized proxy \*/

#endif

);

func->SetIsGlobalFunc(true);

scriptContext->RegisterDynamicFunctionReference(func);

LEAVE\_PINNED\_SCOPE();

return func;

}

/\* static \*/

bool ByteCodeGenerator::NeedScopeObjectForArguments(FuncInfo \*funcInfo, ParseNode \*pnodeFnc)

{

// We can avoid creating a scope object with arguments present if:

bool dontNeedScopeObject =

// We have arguments, and

funcInfo->GetHasHeapArguments()

// Either we are in strict mode, or have strict mode formal semantics from a non-simple parameter list, and

&& (funcInfo->GetIsStrictMode()

|| !pnodeFnc->sxFnc.IsSimpleParameterList())

// Neither of the scopes are objects

&& !funcInfo->paramScope->GetIsObject()

&& !funcInfo->bodyScope->GetIsObject();

return funcInfo->GetHasHeapArguments()

// Regardless of the conditions above, we won't need a scope object if there aren't any formals.

&& (pnodeFnc->sxFnc.pnodeArgs != nullptr || pnodeFnc->sxFnc.pnodeRest != nullptr)

&& !dontNeedScopeObject;

}

Js::FunctionBody \*ByteCodeGenerator::EnsureFakeGlobalFuncForUndefer(ParseNode \*pnode)

{

Js::FunctionBody \*func = scriptContext->GetFakeGlobalFuncForUndefer();

if (!func)

{

func = this->MakeGlobalFunctionBody(pnode);

scriptContext->SetFakeGlobalFuncForUndefer(func);

}

else

{

func->SetBoundPropertyRecords(EnsurePropertyRecordList());

}

if (pnode->sxFnc.GetStrictMode() != 0)

{

func->SetIsStrictMode();

}

return func;

}

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

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

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

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

#if defined(\_M\_ARM32\_OR\_ARM64) || defined(\_M\_X64)

const long AstBytecodeRatioEstimate = 4;

#else

const long AstBytecodeRatioEstimate = 5;

#endif

class ByteCodeGenerator

{

private:

Js::ScriptContext\* scriptContext;

ArenaAllocator \*alloc;

ulong flags;

Js::PropertyRecordList\* propertyRecords;

SList<FuncInfo\*> \*funcInfoStack;

ParseNode \*currentBlock;

ParseNode \*currentTopStatement;

Scope \*currentScope;

Scope \*globalScope; // the global members will be in this scope

Js::ScopeInfo\* parentScopeInfo;

Js::ByteCodeWriter m\_writer;

// pointer to the root function wrapper that will be invoked by the caller

Js::ParseableFunctionInfo \* pRootFunc;

long maxAstSize;

uint16 envDepth;

uint sourceIndex;

uint dynamicScopeCount;

uint loopDepth;

uint16 m\_callSiteId;

bool isBinding;

bool trackEnvDepth;

bool funcEscapes;

bool inPrologue;

bool inDestructuredPattern;

Parser\* parser; // currently active parser (used for AST transformation)

Js::Utf8SourceInfo \*m\_utf8SourceInfo;

// The stack walker won't be able to find the current function being defer parse, pass in

// The address so we can patch it up if it is a stack function and we need to box it.

Js::ScriptFunction \*\* functionRef;

public:

// This points to the current function body which can be reused when parsing a subtree (called due to deferred parsing logic).

Js::FunctionBody \* pCurrentFunction;

bool InDestructuredPattern() const { return inDestructuredPattern; }

void SetInDestructuredPattern(bool in) { inDestructuredPattern = in; }

bool InPrologue() const { return inPrologue; }

void SetInPrologue(bool val) { inPrologue = val; }

Parser\* GetParser() { return parser; }

Js::ParseableFunctionInfo \* GetRootFunc(){return pRootFunc;}

void SetRootFuncInfo(FuncInfo\* funcInfo);

// Treat the return value register like a constant register so that the byte code writer maps it to the bottom

// of the register range.

static const Js::RegSlot ReturnRegister = REGSLOT\_TO\_CONSTREG(Js::FunctionBody::ReturnValueRegSlot);

static const Js::RegSlot RootObjectRegister = REGSLOT\_TO\_CONSTREG(Js::FunctionBody::RootObjectRegSlot);

static const unsigned int DefaultArraySize = 0; // This \_\_must\_\_ be '0' so that "(new Array()).length == 0"

static const unsigned int MinArgumentsForCallOptimization = 16;

bool forceNoNative;

ByteCodeGenerator(Js::ScriptContext\* scriptContext, Js::ScopeInfo\* parentScopeInfo);

#if DBG\_DUMP

bool Trace() const

{

return Js::Configuration::Global.flags.Trace.IsEnabled(Js::ByteCodePhase);

}

#else

bool Trace() const

{

return false;

}

#endif

Js::ScriptContext\* GetScriptContext() { return scriptContext; }

Scope \*GetCurrentScope() const { return currentScope; }

void SetCurrentBlock(ParseNode \*pnode) { currentBlock = pnode; }

ParseNode \*GetCurrentBlock() const { return currentBlock; }

void SetCurrentTopStatement(ParseNode \*pnode) { currentTopStatement = pnode; }

ParseNode \*GetCurrentTopStatement() const { return currentTopStatement; }

Js::ModuleID GetModuleID() const

{

return m\_utf8SourceInfo->GetSrcInfo()->moduleID;

}

void SetFlags(ulong grfscr)

{

flags = grfscr;

}

ulong GetFlags(void)

{

return flags;

}

bool IsConsoleScopeEval(void)

{

return (flags & fscrConsoleScopeEval) == fscrConsoleScopeEval;

}

bool IsBinding() const {

return isBinding;

}

Js::ByteCodeWriter \*Writer() {

return &m\_writer;

}

ArenaAllocator \*GetAllocator() {

return alloc;

}

Js::PropertyRecordList\* EnsurePropertyRecordList()

{

if (this->propertyRecords == nullptr)

{

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

this->propertyRecords = RecyclerNew(recycler, Js::PropertyRecordList, recycler);

}

return this->propertyRecords;

}

bool IsEvalWithBlockScopingNoParentScopeInfo()

{

return (flags & fscrEvalCode) && !HasParentScopeInfo() && scriptContext->GetConfig()->IsBlockScopeEnabled();

}

Js::ProfileId GetNextCallSiteId(Js::OpCode op)

{

if (m\_writer.ShouldIncrementCallSiteId(op))

{

if (m\_callSiteId != Js::Constants::NoProfileId)

{

return m\_callSiteId++;

}

}

return m\_callSiteId;

}

Js::RegSlot NextVarRegister();

Js::RegSlot NextConstRegister();

FuncInfo \*TopFuncInfo() const;

void EnterLoop();

void ExitLoop() { loopDepth--; }

BOOL IsInLoop() const { return loopDepth > 0; }

// TODO: per-function register assignment for env and global symbols

void AssignRegister(Symbol \*sym);

void AddTargetStmt(ParseNode \*pnodeStmt);

Js::RegSlot AssignNullConstRegister();

Js::RegSlot AssignUndefinedConstRegister();

Js::RegSlot AssignTrueConstRegister();

Js::RegSlot AssignFalseConstRegister();

Js::RegSlot AssignThisRegister();

Js::RegSlot AssignNewTargetRegister();

void SetNeedEnvRegister();

void AssignFrameObjRegister();

void AssignFrameSlotsRegister();

void AssignFrameDisplayRegister();

void InitScopeSlotArray(FuncInfo \* funcInfo);

void FinalizeRegisters(FuncInfo \* funcInfo, Js::FunctionBody \* byteCodeFunction);

void SetHasTry(bool has);

void SetHasFinally(bool has);

void SetNumberOfInArgs(Js::ArgSlot argCount);

Js::RegSlot EnregisterConstant(unsigned int constant);

Js::RegSlot EnregisterStringConstant(IdentPtr pid);

Js::RegSlot EnregisterDoubleConstant(double d);

Js::RegSlot EnregisterStringTemplateCallsiteConstant(ParseNode\* pnode);

static Js::JavascriptArray\* BuildArrayFromStringList(ParseNode\* stringNodeList, uint arrayLength, Js::ScriptContext\* scriptContext);

bool HasParentScopeInfo() const

{

return this->parentScopeInfo != nullptr;

}

void RestoreScopeInfo(Js::FunctionBody\* funcInfo);

FuncInfo \*StartBindGlobalStatements(ParseNode \*pnode);

void AssignPropertyId(Symbol \*sym, Js::ParseableFunctionInfo\* functionInfo);

void AssignPropertyId(IdentPtr pid);

void ProcessCapturedSyms(ParseNode \*pnodeFnc);

void RecordAllIntConstants(FuncInfo \* funcInfo);

void RecordAllStrConstants(FuncInfo \* funcInfo);

void RecordAllStringTemplateCallsiteConstants(FuncInfo\* funcInfo);

// For now, this just assigns field ids for the current script.

// Later, we will combine this information with the global field ID map.

// This temporary code will not work if a global member is accessed both with and without a LHS.

void AssignPropertyIds(Js::ParseableFunctionInfo\* functionInfo);

void MapCacheIdsToPropertyIds(FuncInfo \*funcInfo);

void MapReferencedPropertyIds(FuncInfo \*funcInfo);

FuncInfo \*StartBindFunction(const wchar\_t \*name, uint nameLength, uint shortNameOffset, bool\* pfuncExprWithName, ParseNode \*pnode);

void EndBindFunction(bool funcExprWithName);

void StartBindCatch(ParseNode \*pnode);

// Block scopes related functions

template<class Fn> void IterateBlockScopedVariables(ParseNode \*pnodeBlock, Fn fn);

void InitBlockScopedContent(ParseNode \*pnodeBlock, Js::DebuggerScope \*debuggerScope, FuncInfo \*funcInfo);

Js::DebuggerScope\* RecordStartScopeObject(ParseNode \*pnodeBlock, Js::DiagExtraScopesType scopeType, Js::RegSlot scopeLocation = Js::Constants::NoRegister, int\* index = nullptr);

void RecordEndScopeObject(ParseNode \*pnodeBlock);

void EndBindCatch();

void StartEmitFunction(ParseNode \*pnodeFnc);

void EndEmitFunction(ParseNode \*pnodeFnc);

void StartEmitBlock(ParseNode \*pnodeBlock);

void EndEmitBlock(ParseNode \*pnodeBlock);

void StartEmitCatch(ParseNode \*pnodeCatch);

void EndEmitCatch(ParseNode \*pnodeCatch);

void StartEmitWith(ParseNode \*pnodeWith);

void EndEmitWith(ParseNode \*pnodeWith);

void EnsureFncScopeSlots(ParseNode \*pnode, FuncInfo \*funcInfo);

void EnsureLetConstScopeSlots(ParseNode \*pnodeBlock, FuncInfo \*funcInfo);

void PushScope(Scope \*innerScope);

void PopScope();

void PushBlock(ParseNode \*pnode);

void PopBlock();

void PushFuncInfo(wchar\_t const \* location, FuncInfo\* funcInfo);

void PopFuncInfo(wchar\_t const \* location);

Js::RegSlot PrependLocalScopes(Js::RegSlot evalEnv, Js::RegSlot tempLoc, FuncInfo \*funcInfo);

Symbol \*FindSymbol(Symbol \*\*symRef, IdentPtr pid, bool forReference = false);

Symbol \*AddSymbolToScope(Scope \*scope, const wchar\_t \*key, int keyLength, ParseNode \*varDecl, SymbolType symbolType);

Symbol \*AddSymbolToFunctionScope(const wchar\_t \*key, int keyLength, ParseNode \*varDecl, SymbolType symbolType);

void FuncEscapes(Scope \*scope);

void EmitTopLevelStatement(ParseNode \*stmt, FuncInfo \*funcInfo, BOOL fReturnValue);

void EmitInvertedLoop(ParseNode\* outerLoop,ParseNode\* invertedLoop,FuncInfo\* funcInfo);

void DefineFunctions(FuncInfo \*funcInfoParent);

Js::RegSlot DefineOneFunction(ParseNode \*pnodeFnc, FuncInfo \*funcInfoParent, bool generateAssignment=true, Js::RegSlot regEnv = Js::Constants::NoRegister, Js::RegSlot frameDisplayTemp = Js::Constants::NoRegister);

void DefineCachedFunctions(FuncInfo \*funcInfoParent);

void DefineUncachedFunctions(FuncInfo \*funcInfoParent);

void DefineUserVars(FuncInfo \*funcInfo);

void InitBlockScopedNonTemps(ParseNode \*pnode, FuncInfo \*funcInfo);

// temporarily load all constants and special registers in a single block

void LoadAllConstants(FuncInfo \*funcInfo);

void LoadHeapArguments(FuncInfo \*funcInfo);

void LoadUncachedHeapArguments(FuncInfo \*funcInfo);

void LoadCachedHeapArguments(FuncInfo \*funcInfo);

void LoadThisObject(FuncInfo \*funcInfo, bool thisLoadedFromParams = false);

void EmitThis(FuncInfo \*funcInfo, Js::RegSlot fromRegister);

void LoadNewTargetObject(FuncInfo \*funcInfo);

void GetEnclosingNonLambdaScope(FuncInfo \*funcInfo, Scope \* &scope, Js::PropertyId &envIndex);

void EmitInternalScopedSlotLoad(FuncInfo \*funcInfo, Js::RegSlot slot, Js::RegSlot symbolRegister, bool chkUndecl = false);

void EmitInternalScopedSlotLoad(FuncInfo \*funcInfo, Scope \*scope, Js::PropertyId envIndex, Js::RegSlot slot, Js::RegSlot symbolRegister, bool chkUndecl = false);

void EmitInternalScopedSlotStore(FuncInfo \*funcInfo, Js::RegSlot slot, Js::RegSlot symbolRegister);

void EmitInternalScopeObjInit(FuncInfo \*funcInfo, Scope \*scope, Js::RegSlot valueLocation, Js::PropertyId propertyId);

void EmitSuperCall(FuncInfo\* funcInfo, ParseNode\* pnode, BOOL fReturnValue);

void EmitScopeSlotLoadThis(FuncInfo \*funcInfo, Js::RegSlot regLoc, bool chkUndecl = true);

void EmitScopeSlotStoreThis(FuncInfo \*funcInfo, Js::RegSlot regLoc, bool chkUndecl = false);

void EmitClassConstructorEndCode(FuncInfo \*funcInfo);

void EmitBaseClassConstructorThisObject(FuncInfo \*funcInfo);

// TODO: home the 'this' argument

void EmitLoadFormalIntoRegister(ParseNode \*pnodeFormal, Js::RegSlot pos, FuncInfo \*funcInfo);

void HomeArguments(FuncInfo \*funcInfo);

void EnsureNoRedeclarations(ParseNode \*pnodeBlock, FuncInfo \*funcInfo);

void DefineLabels(FuncInfo \*funcInfo);

void EmitProgram(ParseNode \*pnodeProg);

void EmitScopeList(ParseNode \*pnode);

void EmitDefaultArgs(FuncInfo \*funcInfo, ParseNode \*pnode);

void EmitOneFunction(ParseNode \*pnode);

void EmitGlobalFncDeclInit(Js::RegSlot rhsLocation, Js::PropertyId propertyId, FuncInfo \* funcInfo);

void EmitLocalPropInit(Js::RegSlot rhsLocation, Symbol \*sym, FuncInfo \*funcInfo);

void EmitPropStore(Js::RegSlot rhsLocation, Symbol \*sym, IdentPtr pid, FuncInfo \*funcInfo, bool isLet = false, bool isConst = false, bool isFncDeclVar = false);

void EmitPropLoad(Js::RegSlot lhsLocation, Symbol \*sym, IdentPtr pid, FuncInfo \*funcInfo);

void EmitPropDelete(Js::RegSlot lhsLocation, Symbol \*sym, IdentPtr pid, FuncInfo \*funcInfo);

void EmitPropTypeof(Js::RegSlot lhsLocation, Symbol \*sym, IdentPtr pid, FuncInfo \*funcInfo);

void EmitTypeOfFld(FuncInfo \* funcInfo, Js::PropertyId propertyId, Js::RegSlot value, Js::RegSlot instance, Js::OpCode op1);

void EmitLoadInstance(Symbol \*sym, IdentPtr pid, Js::RegSlot \*pThisLocation, Js::RegSlot \*pTargetLocation, FuncInfo \*funcInfo);

void EmitGlobalBody(FuncInfo \*funcInfo);

void EmitFunctionBody(FuncInfo \*funcInfo);

void EmitAsmFunctionBody(FuncInfo \*funcInfo);

void EmitScopeObjectInit(FuncInfo \*funcInfo);

void EmitPatchableRootProperty(Js::OpCode opcode, Js::RegSlot regSlot, Js::PropertyId propertyId, bool isLoadMethod, bool isStore, FuncInfo \*funcInfo);

struct TryScopeRecord;

JsUtil::DoublyLinkedList<TryScopeRecord> tryScopeRecordsList;

void EmitLeaveOpCodesBeforeYield();

void EmitTryBlockHeadersAfterYield();

void InvalidateCachedOuterScopes(FuncInfo \*funcInfo);

bool InDynamicScope() const { return dynamicScopeCount != 0; }

Scope \* FindScopeForSym(Scope \*symScope, Scope \*scope, Js::PropertyId \*envIndex, FuncInfo \*funcInfo) const;

static Js::OpCode GetStFldOpCode(bool isStrictMode, bool isRoot, bool isLetDecl, bool isConstDecl, bool isClassMemberInit)

{

return isClassMemberInit ? Js::OpCode::InitClassMember :

isConstDecl ? (isRoot ? Js::OpCode::InitRootConstFld : Js::OpCode::InitConstFld) :

isLetDecl ? (isRoot ? Js::OpCode::InitRootLetFld : Js::OpCode::InitLetFld) :

isStrictMode ? (isRoot ? Js::OpCode::StRootFldStrict : Js::OpCode::StFldStrict) :

isRoot ? Js::OpCode::StRootFld : Js::OpCode::StFld;

}

static Js::OpCode GetStFldOpCode(FuncInfo\* funcInfo, bool isRoot, bool isLetDecl, bool isConstDecl, bool isClassMemberInit);

static Js::OpCode GetScopedStFldOpCode(bool isStrictMode)

{

return isStrictMode ? Js::OpCode::ScopedStFldStrict : Js::OpCode::ScopedStFld;

}

static Js::OpCode GetScopedStFldOpCode(FuncInfo\* funcInfo, bool isConsoleScopeLetConst = false);

static Js::OpCode GetStElemIOpCode(bool isStrictMode)

{

return isStrictMode ? Js::OpCode::StElemI\_A\_Strict : Js::OpCode::StElemI\_A;

}

static Js::OpCode GetStElemIOpCode(FuncInfo\* funcInfo);

bool DoJitLoopBodies(FuncInfo \*funcInfo) const;

static void Generate(\_\_in ParseNode \*pnode, ulong grfscr, \_\_in ByteCodeGenerator\* byteCodeGenerator, \_\_inout Js::ParseableFunctionInfo \*\* ppRootFunc, \_\_in uint sourceIndex, \_\_in bool forceNoNative, \_\_in Parser\* parser, Js::ScriptFunction \*\* functionRef);

void Begin(

\_\_in ArenaAllocator \*alloc,

\_\_in ulong grfscr,

\_\_in Js::ParseableFunctionInfo\* pRootFunc);

void SetCurrentSourceIndex(uint sourceIndex) { this->sourceIndex = sourceIndex; }

uint GetCurrentSourceIndex() { return sourceIndex; }

static bool IsFalse(ParseNode\* node);

void StartStatement(ParseNode\* node);

void EndStatement(ParseNode\* node);

void StartSubexpression(ParseNode\* node);

void EndSubexpression(ParseNode\* node);

bool UseParserBindings() const;

bool IsES6DestructuringEnabled() const;

bool IsES6ForLoopSemanticsEnabled() const;

// Debugger methods.

bool IsInDebugMode() const;

bool IsInNonDebugMode() const;

bool ShouldTrackDebuggerMetadata() const;

void TrackRegisterPropertyForDebugger(Js::DebuggerScope \*debuggerScope, Symbol \*symbol, FuncInfo \*funcInfo, Js::DebuggerScopePropertyFlags flags = Js::DebuggerScopePropertyFlags\_None, bool isFunctionDeclaration = false);

void TrackActivationObjectPropertyForDebugger(Js::DebuggerScope \*debuggerScope, Symbol \*symbol, Js::DebuggerScopePropertyFlags flags = Js::DebuggerScopePropertyFlags\_None, bool isFunctionDeclaration = false);

void TrackSlotArrayPropertyForDebugger(Js::DebuggerScope \*debuggerScope, Symbol\* symbol, Js::PropertyId propertyId, Js::DebuggerScopePropertyFlags flags = Js::DebuggerScopePropertyFlags\_None, bool isFunctionDeclaration = false);

void TrackFunctionDeclarationPropertyForDebugger(Symbol \*functionDeclarationSymbol, FuncInfo \*funcInfoParent);

void UpdateDebuggerPropertyInitializationOffset(Js::RegSlot location, Js::PropertyId propertyId, bool shouldConsumeRegister = true);

FuncInfo \*FindEnclosingNonLambda();

bool CanStackNestedFunc(FuncInfo \* funcInfo, bool trace = false);

void CheckDeferParseHasMaybeEscapedNestedFunc();

bool NeedObjectAsFunctionScope(FuncInfo \* funcInfo, ParseNode \* pnodeFnc) const;

bool HasInterleavingDynamicScope(Symbol \* sym) const;

void MarkThisUsedInLambda();

void EmitInitCapturedThis(FuncInfo\* funcInfo, Scope\* scope);

void EmitInitCapturedNewTarget(FuncInfo\* funcInfo, Scope\* scope);

Js::FunctionBody \*EnsureFakeGlobalFuncForUndefer(ParseNode \*pnode);

Js::FunctionBody \*MakeGlobalFunctionBody(ParseNode \*pnode);

static bool NeedScopeObjectForArguments(FuncInfo \*funcInfo, ParseNode \*pnodeFnc);

Js::OpCode GetStSlotOp(Scope \*scope, int envIndex, Js::RegSlot scopeLocation, bool chkBlockVar, FuncInfo \*funcInfo);

Js::OpCode GetLdSlotOp(Scope \*scope, int envIndex, Js::RegSlot scopeLocation, FuncInfo \*funcInfo);

Js::OpCode GetInitFldOp(Scope \*scope, Js::RegSlot scopeLocation, FuncInfo \*funcInfo, bool letDecl = false);

private:

bool NeedCheckBlockVar(Symbol\* sym, Scope\* scope, FuncInfo\* funcInfo) const;

Js::OpCode ToChkUndeclOp(Js::OpCode op) const;

};

template<class Fn> void ByteCodeGenerator::IterateBlockScopedVariables(ParseNode \*pnodeBlock, Fn fn)

{

Assert(pnodeBlock->nop == knopBlock);

for (auto lexvar = pnodeBlock->sxBlock.pnodeLexVars; lexvar; lexvar = lexvar->sxVar.pnodeNext)

{

fn(lexvar);

}

}

struct ApplyCheck {

bool matches;

bool insideApplyCall;

bool sawApply;

};

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

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

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

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

#include "RuntimeByteCodePch.h"

namespace Js

{

void ByteCodeReader::Create(FunctionBody \* functionRead, uint startOffset /\* = 0 \*/)

{

Assert(functionRead);

ByteCodeReader::Create(functionRead, startOffset, /\* useOriginalByteCode = \*/ false);

}

void ByteCodeReader::Create(FunctionBody\* functionRead, uint startOffset, bool useOriginalByteCode)

{

AssertMsg(functionRead != nullptr, "Must provide valid function to execute");

ByteBlock \* byteCodeBlock = useOriginalByteCode ?

functionRead->GetOriginalByteCode() :

functionRead->GetByteCode();

AssertMsg(byteCodeBlock != nullptr, "Must have valid byte-code to read");

m\_startLocation = byteCodeBlock->GetBuffer();

m\_currentLocation = m\_startLocation + startOffset;

#if DBG

m\_endLocation = m\_startLocation + byteCodeBlock->GetLength();

Assert(m\_currentLocation <= m\_endLocation);

#endif

}

template<typename LayoutType>

const unaligned LayoutType \* ByteCodeReader::GetLayout()

{

size\_t layoutSize = sizeof(LayoutType);

AssertMsg((layoutSize > 0) && (layoutSize < 100), "Ensure valid layout size");

const byte \* layoutData = m\_currentLocation;

m\_currentLocation += layoutSize;

Assert(m\_currentLocation <= m\_endLocation);

return reinterpret\_cast<const unaligned LayoutType \*>(layoutData);

}

template<typename LayoutType>

const unaligned LayoutType \* ByteCodeReader::GetLayout(const byte\*& ip)

{

size\_t layoutSize = sizeof(LayoutType);

AssertMsg((layoutSize > 0) && (layoutSize < 100), "Ensure valid layout size");

const byte \* layoutData = ip;

ip += layoutSize;

m\_currentLocation = ip;

Assert(m\_currentLocation <= m\_endLocation);

return reinterpret\_cast<const unaligned LayoutType \*>(layoutData);

}

template<>

const unaligned OpLayoutEmpty \* ByteCodeReader::GetLayout<OpLayoutEmpty>()

{

return nullptr;

}

template<>

const unaligned OpLayoutEmpty \* ByteCodeReader::GetLayout<OpLayoutEmpty>(const byte\*& ip)

{

m\_currentLocation = ip;

return nullptr;

}

OpCode ByteCodeReader::ReadOp(const byte \*&ip, LayoutSize& layoutSize) const

{

// Return current location and advance past data.

Assert(ip < m\_endLocation);

OpCode op = (OpCode)\*ip++;

if (!OpCodeUtil::IsPrefixOpcode(op))

{

layoutSize = SmallLayout;

return op;

}

return ReadPrefixedOp(ip, layoutSize, op);

}

OpCode ByteCodeReader::ReadPrefixedOp(const byte \*&ip, LayoutSize& layoutSize, OpCode prefix) const

{

Assert(ip < m\_endLocation);

OpCode op = (OpCode)\*ip++;

switch (prefix)

{

case Js::OpCode::MediumLayoutPrefix:

layoutSize = MediumLayout;

return op;

case Js::OpCode::LargeLayoutPrefix:

layoutSize = LargeLayout;

return op;

case Js::OpCode::ExtendedOpcodePrefix:

layoutSize = SmallLayout;

break;

case Js::OpCode::ExtendedMediumLayoutPrefix:

layoutSize = MediumLayout;

break;

default:

Assert(prefix == Js::OpCode::ExtendedLargeLayoutPrefix);

layoutSize = LargeLayout;

}

return (OpCode)(op + (Js::OpCode::ExtendedOpcodePrefix << 8));

}

OpCode ByteCodeReader::ReadOp(LayoutSize& layoutSize)

{

OpCode op = ReadOp(m\_currentLocation, layoutSize);

#if ENABLE\_NATIVE\_CODEGEN

Assert(!OpCodeAttr::BackEndOnly(op));

#endif

return op;

}

OpCode ByteCodeReader::ReadPrefixedOp(LayoutSize& layoutSize, OpCode prefix)

{

Assert(OpCodeUtil::IsPrefixOpcode(prefix));

return ReadPrefixedOp(m\_currentLocation, layoutSize, prefix);

}

OpCode ByteCodeReader::PeekOp(LayoutSize& layoutSize) const

{

const byte \* ip = m\_currentLocation;

return ReadOp(ip, layoutSize);

}

OpCode ByteCodeReader::PeekOp(const byte \* ip, LayoutSize& layoutSize)

{

return ReadOp(ip, layoutSize);

}

OpCode ByteCodeReader::ReadByteOp(const byte\*& ip)

{

return (OpCode)\*ip++;

}

OpCode ByteCodeReader::PeekByteOp(const byte \* ip)

{

return (OpCode)\*ip;

}

const byte\* ByteCodeReader::GetIP()

{

return m\_currentLocation;

}

void ByteCodeReader::SetIP(const byte \*const ip)

{

Assert(ip >= m\_startLocation);

Assert(ip < m\_endLocation);

m\_currentLocation = ip;

}

// Define reading functions

#define LAYOUT\_TYPE(layout) \

const unaligned OpLayout##layout \* ByteCodeReader::layout() \

{ \

return GetLayout<OpLayout##layout>(); \

} \

const unaligned OpLayout##layout \* ByteCodeReader::layout(const byte\*& ip) \

{ \

return GetLayout<OpLayout##layout>(ip); \

}

#include "LayoutTypes.h"

// Define reading functions

#define LAYOUT\_TYPE(layout) \

const unaligned OpLayout##layout \* ByteCodeReader::layout() \

{ \

return GetLayout<OpLayout##layout>(); \

} \

const unaligned OpLayout##layout \* ByteCodeReader::layout(const byte\*& ip) \

{ \

return GetLayout<OpLayout##layout>(ip); \

}

#define EXCLUDE\_DUP\_LAYOUT

#include "LayoutTypesAsmJs.h"

uint ByteCodeReader::GetCurrentOffset() const

{

Assert(m\_currentLocation >= m\_startLocation);

Assert(m\_currentLocation - m\_startLocation <= UINT\_MAX);

return (uint)(m\_currentLocation - m\_startLocation);

}

const byte \* ByteCodeReader::SetCurrentOffset(int byteOffset)

{

const byte \* ip = m\_startLocation + byteOffset;

Assert(ip < m\_endLocation);

m\_currentLocation = ip;

return ip;

}

const byte \* ByteCodeReader::SetCurrentRelativeOffset(const byte \* ip, int byteOffset)

{

Assert(ip < m\_endLocation);

const byte \* targetip = ip + byteOffset;

Assert(targetip < m\_endLocation);

m\_currentLocation = targetip;

return targetip;

}

template <typename T>

AuxArray<T> const \* ByteCodeReader::ReadAuxArray(uint offset, FunctionBody \* functionBody)

{

Js::AuxArray<T> const \* auxArray = (Js::AuxArray<T> const \*)(functionBody->GetAuxiliaryData()->GetBuffer() + offset);

Assert(offset + auxArray->GetDataSize() <= functionBody->GetAuxiliaryData()->GetLength());

return auxArray;

}

// explicit instantiations

template AuxArray<Var> const \* ByteCodeReader::ReadAuxArray<Var>(uint offset, FunctionBody \* functionBody);

template AuxArray<int32> const \* ByteCodeReader::ReadAuxArray<int32>(uint offset, FunctionBody \* functionBody);

template AuxArray<uint32> const \* ByteCodeReader::ReadAuxArray<uint32>(uint offset, FunctionBody \* functionBody);

template AuxArray<double> const \* ByteCodeReader::ReadAuxArray<double>(uint offset, FunctionBody \* functionBody);

template AuxArray<FuncInfoEntry> const \* ByteCodeReader::ReadAuxArray<FuncInfoEntry>(uint offset, FunctionBody \* functionBody);

const Js::PropertyIdArray \* ByteCodeReader::ReadPropertyIdArray(uint offset, FunctionBody \* functionBody, uint extraSlots)

{

Js::PropertyIdArray const \* propIds = (Js::PropertyIdArray const \*)(functionBody->GetAuxiliaryData()->GetBuffer() + offset);

Assert(offset + propIds->GetDataSize(extraSlots) <= functionBody->GetAuxiliaryData()->GetLength());

return propIds;

}

size\_t VarArrayVarCount::GetDataSize() const

{

return sizeof(VarArrayVarCount) + sizeof(Var) \* TaggedInt::ToInt32(count);

}

void VarArrayVarCount::SetCount(uint count)

{

this->count = Js::TaggedInt::ToVarUnchecked(count);

}

const Js::VarArrayVarCount \* ByteCodeReader::ReadVarArrayVarCount(uint offset, FunctionBody \* functionBody)

{

Js::VarArrayVarCount const \* varArray = (Js::VarArrayVarCount const \*)(functionBody->GetAuxiliaryContextData()->GetBuffer() + offset);

Assert(offset + varArray->GetDataSize() <= functionBody->GetAuxiliaryContextData()->GetLength());

return varArray;

}

#if DBG\_DUMP

byte ByteCodeReader::GetRawByte(int i)

{

return m\_startLocation[i];

}

#endif

} // 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.

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

#pragma once

namespace Js

{

struct ByteCodeReader

{

static uint32 GetStartLocationOffset() { return offsetof(ByteCodeReader, m\_startLocation); }

static uint32 GetCurrentLocationOffset() { return offsetof(ByteCodeReader, m\_currentLocation); }

private:

const byte \* m\_startLocation;

const byte \* m\_currentLocation;

#if DBG

const byte \* m\_endLocation;

#endif

public:

void Create(FunctionBody\* functionRead, uint startOffset = 0);

void Create(FunctionBody\* functionRead, uint startOffset, bool useOriginalByteCode);

uint GetCurrentOffset() const;

const byte \* SetCurrentOffset(int byteOffset);

const byte \* SetCurrentRelativeOffset(const byte \* ip, int byteOffset);

template<typename LayoutType> inline const unaligned LayoutType \* GetLayout();

template<typename LayoutType> inline const unaligned LayoutType \* GetLayout(const byte\*& ip);

// Read\*Op advance the IP,

// Peek\*Op doesn't move the IP

// \*ByteOp only read one byte of the opcode,

// \*Op interprets and remove the large layout prefix

private:

OpCode ReadOp(const byte \*&ip, LayoutSize& layoutSize) const;

OpCode ReadPrefixedOp(const byte \*&ip, LayoutSize& layoutSize, OpCode prefix) const;

public:

OpCode ReadOp(LayoutSize& layoutSize);

OpCode ReadPrefixedOp(LayoutSize& layoutSize, OpCode prefix);

OpCode PeekOp(LayoutSize& layoutSize) const;

OpCode PeekOp() const { LayoutSize layoutSize; return PeekOp(layoutSize); }

OpCode PeekOp(const byte \* ip, LayoutSize& layoutSize);

static OpCode ReadByteOp(const byte\*& ip);

static OpCode PeekByteOp(const byte \* ip);

// Declare reading functions

#define LAYOUT\_TYPE(layout) \

const unaligned OpLayout##layout\* layout(); \

const unaligned OpLayout##layout\* layout(const byte\*& ip);

#include "LayoutTypes.h"

#ifndef TEMP\_DISABLE\_ASMJS

#define LAYOUT\_TYPE(layout) \

const unaligned OpLayout##layout\* layout(); \

const unaligned OpLayout##layout\* layout(const byte\*& ip);

#define EXCLUDE\_DUP\_LAYOUT

#include "LayoutTypesAsmJs.h"

#endif

template <typename T>

static AuxArray<T> const \* ReadAuxArray(uint offset, FunctionBody \* functionBody);

static PropertyIdArray const \* ReadPropertyIdArray(uint offset, FunctionBody \* functionBody, uint extraSlots = 0);

static VarArrayVarCount const \* ReadVarArrayVarCount(uint offset, FunctionBody \* functionBody);

const byte\* GetIP();

void SetIP(const byte \*const ip);

template<class T> const unaligned T\* AuxiliaryContext(const byte\*& ip, const byte \*\* content);

#if DBG\_DUMP

byte GetRawByte(int i);

#endif

};

} // 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.

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

#pragma once

#define GENERATE\_BYTE\_CODE\_BUFFER\_LIBRARY 0x00000001

#define GENERATE\_BYTE\_CODE\_FOR\_NATIVE 0x00000002

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

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

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

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

#include "RuntimeByteCodePch.h"

#include "RegexCommon.h"

#include "RegexPattern.h"

#include "Library\RegexHelper.h"

#include "DataStructures\Option.h"

#include "DataStructures\ImmutableList.h"

#include "DataStructures\BufferBuilder.h"

#include "ByteCode\OpCodeUtilAsmJs.h"

#include "ByteCode\ByteCodeSerializer.h"

#include "Language\AsmJSModule.h"

#include "Library\ES5Array.h"

void ChakraBinaryBuildDateTimeHash(DWORD \* buildDateHash, DWORD \* buildTimeHash);

namespace Js

{

const int magicConstant = \*(int\*)"ChBc";

const int majorVersionConstant = 1;

const int minorVersionConstant = 1;

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

// These magic constants can be enabled to bracket and check different sections of the serialization

// file. Turn on BYTE\_CODE\_MAGIC\_CONSTANTS in ByteCodeSerializer.h to enable this.

const int magicStartOfFunctionBody = \*(int\*)"fun[";

const int magicEndOfFunctionBody = \*(int\*)"]fun";

const int magicStartOfConstantTable = \*(int\*)"con[";

const int magicEndOfConstantTable = \*(int\*)"]con";

const int magicStartStringConstant = \*(int\*)"str[";

const int magicEndStringConstant = \*(int\*)"]str";

const int magicStartOfCacheIdToPropIdMap = \*(int\*)"cid[";

const int magicEndOfCacheIdToPropIdMap = \*(int\*)"]cid";

const int magicStartOfReferencedPropIdMap = \*(int\*)"rid[";

const int magicEndOfReferencedPropIdMap = \*(int\*)"]rid";

const int magicStartOfPropertyIdsForScopeSlotArray = \*(int\*)"scope[";

const int magicEndOfPropertyIdsForScopeSlotArray = \*(int\*)"]scope";

const int magicStartOfDebuggerScopes = \*(int\*)"dbgscope[";

const int magicEndOfDebuggerScopes = \*(int\*)"]dbgscope";

const int magicStartOfDebuggerScopeProperties = \*(int\*)"dbgscopeprop[";

const int magicEndOfDebuggerScopeProperties = \*(int\*)"]dbgscopeprop";

const int magicStartOfAux = \*(int\*)"aux[";

const int magicEndOfAux = \*(int\*)"]aux";

const int magicStartOfAuxVarArray = \*(int\*)"ava[";

const int magicEndOfAuxVarArray = \*(int\*)"]ava";

const int magicStartOfAuxIntArray = \*(int\*)"aia[";

const int magicEndOfAuxIntArray = \*(int\*)"]aia";

const int magicStartOfAuxFltArray = \*(int\*)"afa[";

const int magicEndOfAuxFltArray = \*(int\*)"]afa";

const int magicStartOfAuxPropIdArray = \*(int\*)"api[";

const int magicEndOfAuxPropIdArray = \*(int\*)"]api";

const int magicStartOfAuxFuncInfoArray = \*(int\*)"afi[";

const int magicEndOfAuxFuncInfoArray = \*(int\*)"]afi";

const int magicStartOfAsmJsFuncInfo = \*(int\*)"asmfuncinfo[";

const int magicEndOfAsmJsFuncInfo = \*(int\*)"]asmfuncinfo";

const int magicStartOfAsmJsModuleInfo = \*(int\*)"asmmodinfo[";

const int magicEndOfAsmJsModuleInfo = \*(int\*)"]asmmodinfo";

#endif

// Serialized files are architecture specific

#ifndef VALIDATE\_SERIALIZED\_BYTECODE

#if \_M\_AMD64

const byte magicArchitecture = 64;

#else

const byte magicArchitecture = 32;

#endif

#else

#if \_M\_AMD64

const int magicArchitecture = \*(int\*)"amd";

#elif \_M\_IA64

const int magicArchitecture = \*(int\*)"ia64";

#elif \_M\_ARM

const int magicArchitecture = \*(int\*)"arm";

#elif \_M\_ARM\_64

const int magicArchitecture = \*(int\*)"arm64";

#else

const int magicArchitecture = \*(int\*)"x86";

#endif

#endif

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

// Byte Code Serializer Versioning scheme

// Version number is a GUID (128 bits). There are two versioning modes--Engineering and Release. Engineering mode is for day-to-day development. Every time chakra.dll is built a

// fresh new version is generated by hashing the build date and time. This means that a byte code file saved to disk is exactly tied to the binary that generated it. This works

// well for QA test runs and buddy tests because there is no chance of effects between runs.

//

// Release mode is used when chakra.dll is close to public release where there are actual changes to chakra. The GUID is a fixed number from build-to-build. This number will stay

// the same for releases where there is no change to chakra.dll. The reason for this is that we don't want to invalidate compatible byte code that has already been cached.

enum FileVersionScheme : byte

{

// Currently Chakra and ChakraCore versioning scheme is different.

// Same version number for Chakra and ChakraCore doesn't mean they are the same.

// Give the versioning scheme different value, so that byte code generate from one won't be use in the other.

#ifdef NTBUILD

EngineeringVersioningScheme = 10,

ReleaseVersioningScheme = 20,

#else

EngineeringVersioningScheme = 11,

ReleaseVersioningScheme = 21,

#endif

#if (defined(NTBUILD) && CHAKRA\_VERSION\_RELEASE) || (!defined(NTBUILD) && CHAKRA\_CORE\_VERSION\_RELEASE)

CurrentFileVersionScheme = ReleaseVersioningScheme

#else

CurrentFileVersionScheme = EngineeringVersioningScheme

#endif

};

// it should be in separate file for testing

#include "byteCodeCacheReleaseFileVersion.h"

// Used for selective serialization of Function Body fields to make the representation compact

#define DECLARE\_SERIALIZABLE\_FIELD(type, name, serializableType) bool has\_##name : 1

#define DECLARE\_MANUAL\_SERIALIZABLE\_FIELD(type, name, serializableType, deserializeHere) bool has\_##name: 1

#define DEFINE\_ALL\_FIELDS

struct SerializedFieldList {

#include "SerializableFunctionFields.h"

bool has\_m\_lineNumber: 1;

bool has\_m\_columnNumber: 1;

};

C\_ASSERT(sizeof(GUID)==sizeof(DWORD)\*4);

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

// Holds a buffer and size for use by the serializer

struct ByteBuffer

{

uint32 byteCount;

union

{

void \* pv;

const wchar\_t \* s16;

const char \* s8;

};

public:

ByteBuffer(uint32 byteCount, void \* pv)

: byteCount(byteCount), pv(pv)

{ }

};

template<>

struct DefaultComparer<ByteBuffer\*>

{

static bool Equals(ByteBuffer const \* str1, ByteBuffer const \* str2)

{

if (str1->byteCount != str2->byteCount)

{

return false;

}

return memcmp(str1->pv, str2->pv, str1->byteCount)==0;

}

static hash\_t GetHashCode(ByteBuffer const \* str)

{

return JsUtil::CharacterBuffer<char>::StaticGetHashCode(str->s8, str->byteCount);

}

};

struct IndexEntry

{

BufferBuilderByte\* isProprertyRecord;

int id;

};

#pragma pack(push, 1)

struct StringIndexRecord

{

int offset;

bool isProprertyRecord;

};

#pragma pack(pop)

typedef JsUtil::BaseDictionary<ByteBuffer\*, IndexEntry, ArenaAllocator, PrimeSizePolicy, DefaultComparer> TString16ToId;

// Boolean flags on the FunctionBody

enum FunctionFlags

{

ffIsDeclaration = 0x0001,

ffHasImplicitArgsIn = 0x0002,

ffIsAccessor = 0x0004,

ffIsGlobalFunc = 0x0008,

ffDontInline = 0x0010,

ffIsFuncRegistered = 0x0020,

ffIsStaticNameFunction = 0x0040,

ffIsStrictMode = 0x0080,

ffDoBackendArgumentsOptimization = 0x0100,

ffIsEval = 0x0200,

ffIsDynamicFunction = 0x0400,

ffhasAllNonLocalReferenced = 0x0800,

ffhasSetIsObject = 0x1000,

ffhasSetCallsEval = 0x2000,

ffIsNameIdentifierRef = 0x4000,

ffChildCallsEval = 0x8000,

ffHasReferenceableBuiltInArguments = 0x10000,

ffIsNamedFunctionExpression = 0x20000,

ffIsAsmJsMode = 0x40000,

ffIsAsmJsFunction = 0x80000

};

// Kinds of constant

enum ConstantType : byte

{

ctInt = 1,

ctString16 = 2,

ctNull = 3,

ctUndefined = 4,

ctNumber = 5,

ctNullDisplay = 6,

ctStrictNullDisplay = 7,

ctTrue = 8,

ctFalse = 9,

ctStringTemplateCallsite = 10

};

// Try to convert from size\_t to uint32. May overflow (and return false) on 64-bit.

bool TryConvertToUInt32(size\_t size, uint32 \* out)

{

\*out = (uint32)size;

if (sizeof(size) == sizeof(uint32))

{

return true;

}

Assert(sizeof(size\_t) == sizeof(uint64));

if((uint64)(\*out) == size)

{

return true;

}

AssertMsg(false, "Is it really an offset greater than 32 bits?"); // More likely a bug somewhere.

return false;

}

#if VARIABLE\_INT\_ENCODING

template <typename T>

static const byte \* ReadVariableInt(const byte \* buffer, size\_t remainingBytes, T \* value)

{

Assert(remainingBytes >= sizeof(byte));

byte firstByte = \*(byte\*) buffer;

if (firstByte >= MIN\_SENTINEL)

{

Assert(remainingBytes >= sizeof(uint16));

const byte\* locationOfValue = buffer + 1;

if (firstByte == TWO\_BYTE\_SENTINEL)

{

uint16 twoByteValue = \*((serialization\_alignment uint16\*) locationOfValue);

Assert(twoByteValue > ONE\_BYTE\_MAX);

\*value = twoByteValue;

PHASE\_PRINT\_TESTTRACE1(Js::VariableIntEncodingPhase, L"TestTrace: VariableIntEncoding (decode)- 2 bytes, value %u\n", \*value);

return buffer + sizeof(uint16) +SENTINEL\_BYTE\_COUNT;

}

else

{

Assert(remainingBytes >= sizeof(T));

Assert(firstByte == FOUR\_BYTE\_SENTINEL);

\*value = \*((serialization\_alignment T\*) locationOfValue);

Assert(\*value > TWO\_BYTE\_MAX || \*value <= 0);

PHASE\_PRINT\_TESTTRACE1(Js::VariableIntEncodingPhase, L"TestTrace: VariableIntEncoding (decode) - 4 bytes, value %u\n", \*value);

return buffer + sizeof(T) +SENTINEL\_BYTE\_COUNT;

}

}

else

{

\*value = (T) firstByte;

PHASE\_PRINT\_TESTTRACE1(Js::VariableIntEncodingPhase, L"TestTrace: VariableIntEncoding (decode) - 1 byte, value %u\n", \*value);

return buffer + sizeof(byte);

}

}

#endif

// Compile-time-check some invariants that the file format depends on

C\_ASSERT(sizeof(PropertyId)==sizeof(int32));

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

// Byte Code File Header Layout

// Offset Size Name Value

// 0 4 Magic Number "ChBc"

// 4 4 Total File Size

// 8 1 File Version Scheme 10 for engineering 20 for release

// 9 4 Version DWORD 1 jscript minor version GUID quad part 1

// 13 4 Version DWORD 2 jscript major version GUID quad part 2

// 17 4 Version DWORD 3 hash of \_\_DATE\_\_ GUID quad part 3

// 21 4 Version DWORD 4 hash of \_\_TIME\_\_ GUID quad part 4

// 25 4 Expected Architecture "amd"0, "ia64", "arm"0 or "x86"0

// 29 4 Expected Function Body Size

// 33 4 Expected Built In PropertyCount

// 37 4 Expected Op Code Count

// 41 4 Size of Original Source Code

// 45 4 Count of Auxiliary Structures

// 49 4 Smallest Literal Object ID

// 53 4 Largest Literal Object ID

// 57 4 Offset from start of this file

// to Strings Table

// 61 4 Offset to Source Spans

// 65 4 Count of Functions

// 69 4 Offset to Functions

// 73 4 Offset to Auxiliary Structures

// 77 4 Count of Strings

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

// This is the serializer

class ByteCodeBufferBuilder

{

// Begin File Layout -------------------------------

ConstantSizedBufferBuilderOf<int32> magic;

ConstantSizedBufferBuilderOf<int32> totalSize; // The size is unknown when the offsets are calculated so just reserve 4 bytes for this for now to avoid doing two passes to calculate the offsets

BufferBuilderByte fileVersionKind; // Engineering or Release

ConstantSizedBufferBuilderOf<int32> V1; // V1-V4 are the parts of the version. It is a fixed version GUID or a per-build version.

ConstantSizedBufferBuilderOf<int32> V2;

ConstantSizedBufferBuilderOf<int32> V3;

ConstantSizedBufferBuilderOf<int32> V4;

BufferBuilderInt32 architecture;

BufferBuilderInt32 expectedFunctionBodySize;

BufferBuilderInt32 expectedBuildInPropertyCount;

BufferBuilderInt32 expectedOpCodeCount;

BufferBuilderInt32 originalSourceSize;

BufferBuilderInt32 originalCharLength;

BufferBuilderRelativeOffset string16sOffset;

BufferBuilderRelativeOffset sourceSpansOffset;

BufferBuilderRelativeOffset lineInfoCacheOffset;

BufferBuilderRelativeOffset functionsOffset;

BufferBuilderInt32 string16Count;

BufferBuilderList string16IndexTable;

BufferBuilderList string16Table;

BufferBuilderAligned alignedString16Table;

BufferBuilderList sourceSpans;

BufferBuilderInt32 lineInfoCacheCount;

BufferBuilderRaw lineInfoCache;

BufferBuilderInt32 functionCount;

BufferBuilderList functionsTable;

// End File Layout ---------------------------------

ArenaAllocator \* alloc;

TString16ToId \* string16ToId;

int nextString16Id;

int topFunctionId;

LPCUTF8 utf8Source;

ScriptContext \* scriptContext;

BufferBuilder \* startOfCachedScopeAuxBlock;

DWORD dwFlags;

DWORD dwFunctionTableLength;

BYTE \*functionTable;

//Instead of referencing TotalNumberOfBuiltInProperties directly; or PropertyIds::\_countJSOnlyProperty we use this.

//For library code this will be set to \_countJSOnlyProperty and for normal bytecode this will be TotalNumberOfBuiltInProperties

int builtInPropertyCount;

bool GenerateLibraryByteCode() const

{

return (dwFlags & GENERATE\_BYTE\_CODE\_BUFFER\_LIBRARY) != 0;

}

bool OmitFunction(int serializationIndex) const

{

return functionTable && ((serializationIndex >= (int)dwFunctionTableLength) || !functionTable[serializationIndex]);

}

bool GenerateByteCodeForNative(int serializationIndex) const

{

return (dwFlags & GENERATE\_BYTE\_CODE\_FOR\_NATIVE) != 0;

}

public:

ByteCodeBufferBuilder(uint32 sourceSize, uint32 sourceCharLength, LPCUTF8 utf8Source, DWORD dwFunctionTableLength, BYTE \* functionTable, Utf8SourceInfo\* sourceInfo, ScriptContext \* scriptContext, ArenaAllocator \* alloc, DWORD dwFlags, int builtInPropertyCount)

: magic(L"Magic", magicConstant),

totalSize(L"Total Size", 0),

fileVersionKind(L"FileVersionKind", 0),

V1(L"V1", 0),

V2(L"V2", 0),

V3(L"V3", 0),

V4(L"V4", 0),

architecture(L"Expected Architecture", magicArchitecture),

expectedFunctionBodySize(L"Expected Function Body Size", sizeof(unaligned FunctionBody)),

expectedBuildInPropertyCount(L"Expected Built-in Properties", builtInPropertyCount),

expectedOpCodeCount(L"Expected Number of OpCodes", (int)OpCode::Count),

originalSourceSize(L"Source Size", sourceSize),

originalCharLength(L"Source Char Length", sourceCharLength),

string16sOffset(L"Offset of String16s", &string16Count),

sourceSpansOffset(L"Offset of Source Spans", &sourceSpans),

lineInfoCacheOffset(L"Offset of Source Spans", &lineInfoCacheCount),

functionCount(L"Function Count", 0),

functionsOffset(L"Offset of Functions", &functionCount),

string16Count(L"String16 Count", 0),

string16IndexTable(L"String16 Indexes"),

string16Table(L"String16 Table"),

alignedString16Table(L"Alignment for String16 Table", &string16Table, sizeof(wchar\_t)),

sourceSpans(L"Source Spans"),

lineInfoCacheCount(L"Line Info Cache", sourceInfo->GetLineOffsetCache()->GetLineCount()),

lineInfoCache(L"Line Info Cache", lineInfoCacheCount.value \* sizeof(JsUtil::LineOffsetCache<Recycler>::LineOffsetCacheItem), (byte \*)sourceInfo->GetLineOffsetCache()->GetItems()),

functionsTable(L"Functions"),

nextString16Id(builtInPropertyCount), // Reserve the built-in property ids

topFunctionId(0),

utf8Source(utf8Source),

scriptContext(scriptContext),

startOfCachedScopeAuxBlock(nullptr),

alloc(alloc),

dwFlags(dwFlags),

builtInPropertyCount(builtInPropertyCount),

dwFunctionTableLength(dwFunctionTableLength),

functionTable(functionTable)

{

if (GenerateLibraryByteCode())

{

expectedFunctionBodySize.value = 0;

expectedOpCodeCount.value = 0;

}

// library alaways use the release versioning scheme

byte actualFileVersionScheme = GenerateLibraryByteCode()? ReleaseVersioningScheme : CurrentFileVersionScheme;

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

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

{

actualFileVersionScheme = (byte)Js::Configuration::Global.flags.ForceSerializedBytecodeVersionSchema;

}

#endif

fileVersionKind.value = actualFileVersionScheme;

if (actualFileVersionScheme != ReleaseVersioningScheme)

{

Assert(!GenerateLibraryByteCode());

Assert(actualFileVersionScheme == EngineeringVersioningScheme);

DWORD jscriptMajor, jscriptMinor, buildDateHash, buildTimeHash;

Js::VerifyOkCatastrophic(AutoSystemInfo::GetJscriptFileVersion(&jscriptMajor, &jscriptMinor, &buildDateHash, &buildTimeHash));

V1.value = jscriptMajor;

V2.value = jscriptMinor;

V3.value = buildDateHash;

V4.value = buildTimeHash;

}

else

{

auto guidDWORDs = (DWORD\*)(&byteCodeCacheReleaseFileVersion);

V1.value = guidDWORDs[0];

V2.value = guidDWORDs[1];

V3.value = guidDWORDs[2];

V4.value = guidDWORDs[3];

}

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

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

{

V1.value = Js::Configuration::Global.flags.ForceSerializedBytecodeMajorVersion;

V2.value = 0;

V3.value = 0;

V4.value = 0;

}

#endif

string16ToId = Anew(alloc, TString16ToId, alloc);

}

HRESULT Create(bool allocateBuffer, byte \*\* buffer, DWORD \* bufferBytes)

{

BufferBuilderList all(L"Final");

// Reverse the lists

string16IndexTable.list = string16IndexTable.list->ReverseCurrentList();

string16Table.list = string16Table.list->ReverseCurrentList();

// Prepend all sections (in reverse order because of prepend)

all.list = all.list->Prepend(&functionsTable, alloc);

all.list = all.list->Prepend(&functionCount, alloc);

all.list = all.list->Prepend(&lineInfoCache, alloc);

all.list = all.list->Prepend(&lineInfoCacheCount, alloc);

all.list = all.list->Prepend(&alignedString16Table, alloc);

all.list = all.list->Prepend(&string16IndexTable, alloc);

all.list = all.list->Prepend(&string16Count, alloc);

all.list = all.list->Prepend(&functionsOffset, alloc);

all.list = all.list->Prepend(&sourceSpansOffset, alloc);

all.list = all.list->Prepend(&lineInfoCacheOffset, alloc);

all.list = all.list->Prepend(&string16sOffset, alloc);

all.list = all.list->Prepend(&originalCharLength, alloc);

all.list = all.list->Prepend(&originalSourceSize, alloc);

all.list = all.list->Prepend(&expectedOpCodeCount, alloc);

all.list = all.list->Prepend(&expectedBuildInPropertyCount, alloc);

all.list = all.list->Prepend(&expectedFunctionBodySize, alloc);

all.list = all.list->Prepend(&architecture, alloc);

all.list = all.list->Prepend(&V4, alloc);

all.list = all.list->Prepend(&V3, alloc);

all.list = all.list->Prepend(&V2, alloc);

all.list = all.list->Prepend(&V1, alloc);

all.list = all.list->Prepend(&fileVersionKind, alloc);

all.list = all.list->Prepend(&totalSize, alloc);

all.list = all.list->Prepend(&magic, alloc);

// Get the string count.

string16Count.value = nextString16Id - this->builtInPropertyCount;

// Figure out the size and set all individual offsets

DWORD size = all.FixOffset(0);

totalSize.value = size;

// Allocate the bytes

if (allocateBuffer)

{

\*bufferBytes = size;

\*buffer = (byte\*)CoTaskMemAlloc(\*bufferBytes);

if (\*buffer == nullptr)

{

return E\_OUTOFMEMORY;

}

}

if (size > \*bufferBytes)

{

\*bufferBytes = size;

return \*buffer == nullptr ? S\_OK : E\_INVALIDARG;

}

else

{

// Write into the buffer

all.Write(\*buffer, \*bufferBytes);

\*bufferBytes = size;

DebugOnly(Output::Flush()); // Flush trace

return S\_OK;

}

}

bool isBuiltinProperty(PropertyId pid) {

if (pid < this->builtInPropertyCount || pid==/\*nil\*/0xffffffff)

{

return true;

}

return false;

};

PropertyId encodeNonBuiltinPropertyId(PropertyId id) {

const PropertyRecord \* propertyValue = nullptr;

Assert(id >= this->builtInPropertyCount); // Shouldn't have gotten a builtin property id

propertyValue = scriptContext->GetPropertyName(id);

id = GetIdOfPropertyRecord(propertyValue) - this->builtInPropertyCount;

return id ^ SERIALIZER\_OBSCURE\_NONBUILTIN\_PROPERTY\_ID;

};

PropertyId encodePossiblyBuiltInPropertyId(PropertyId id) {

const PropertyRecord \* propertyValue = nullptr;

if(id >= this->builtInPropertyCount)

{

propertyValue = scriptContext->GetPropertyName(id);

id = GetIdOfPropertyRecord(propertyValue);

}

return id ^ SERIALIZER\_OBSCURE\_PROPERTY\_ID;

};

int GetString16Id(ByteBuffer \* bb, bool isPropertyRecord = false)

{

IndexEntry indexEntry;

if (!string16ToId->TryGetValue(bb, &indexEntry))

{

auto sizeInBytes = bb->byteCount;

auto stringEntry = Anew(alloc, BufferBuilderRaw, L"String16", sizeInBytes, (const byte \*)bb->pv); // Writing the terminator even though it is computable so that this memory can be used as-is when deserialized

string16Table.list = string16Table.list->Prepend(stringEntry, alloc);

if (string16IndexTable.list == nullptr)

{

// First item in the list is the first string.

auto stringIndexEntry = Anew(alloc, BufferBuilderRelativeOffset, L"First String16 Index", stringEntry);

string16IndexTable.list = string16IndexTable.list->Prepend(stringIndexEntry, alloc);

PrependByte(string16IndexTable, L"isPropertyRecord", (BYTE)isPropertyRecord);

}

// Get a pointer to the previous entry of isPropertyRecord

indexEntry.isProprertyRecord = static\_cast<BufferBuilderByte\*>(string16IndexTable.list->First());

// Subsequent strings indexes point one past the end. This way, the size is always computable by subtracting indexes.

auto stringIndexEntry = Anew(alloc, BufferBuilderRelativeOffset, L"String16 Index", stringEntry, sizeInBytes);

string16IndexTable.list = string16IndexTable.list->Prepend(stringIndexEntry, alloc);

// By default, mark the next string to be not a property record.

PrependByte(string16IndexTable, L"isPropertyRecord", (BYTE)false);

indexEntry.id = nextString16Id;

string16ToId->Add(bb, indexEntry);

++nextString16Id;

}

// A string might start off as not being a property record and later becoming one. Hence,

// we set only if the transition is from false => true. Once it is a property record, it cannot go back.

if(isPropertyRecord)

{

indexEntry.isProprertyRecord->value = isPropertyRecord;

}

return indexEntry.id;

}

uint32 PrependRelativeOffset(BufferBuilderList & builder, LPCWSTR clue, BufferBuilder \* pointedTo)

{

auto entry = Anew(alloc, BufferBuilderRelativeOffset, clue, pointedTo, 0);

builder.list = builder.list->Prepend(entry, alloc);

return sizeof(int32);

}

uint32 PrependInt16(BufferBuilderList & builder, LPCWSTR clue, int16 value, BufferBuilderInt16 \*\* entryOut = nullptr)

{

auto entry = Anew(alloc, BufferBuilderInt16, clue, value);

builder.list = builder.list->Prepend(entry, alloc);

if (entryOut)

{

\*entryOut = entry;

}

return sizeof(int16);

}

uint32 PrependInt32(BufferBuilderList & builder, LPCWSTR clue, int value, BufferBuilderInt32 \*\* entryOut = nullptr)

{

auto entry = Anew(alloc, BufferBuilderInt32, clue, value);

builder.list = builder.list->Prepend(entry, alloc);

if (entryOut)

{

\*entryOut = entry;

}

return sizeof(int32);

}

uint32 PrependConstantInt16(BufferBuilderList & builder, LPCWSTR clue, int16 value, ConstantSizedBufferBuilderOf<int16> \*\* entryOut = nullptr)

{

auto entry = Anew(alloc, ConstantSizedBufferBuilderOf<int16>, clue, value);

builder.list = builder.list->Prepend(entry, alloc);

if (entryOut)

{

\*entryOut = entry;

}

return sizeof(int16);

}

uint32 PrependConstantInt32(BufferBuilderList & builder, LPCWSTR clue, int value, ConstantSizedBufferBuilderOf<int> \*\* entryOut = nullptr)

{

auto entry = Anew(alloc, ConstantSizedBufferBuilderOf<int>, clue, value);

builder.list = builder.list->Prepend(entry, alloc);

if (entryOut)

{

\*entryOut = entry;

}

return sizeof(int32);

}

uint32 PrependConstantInt64(BufferBuilderList & builder, LPCWSTR clue, int64 value, ConstantSizedBufferBuilderOf<int64> \*\* entryOut = nullptr)

{

auto entry = Anew(alloc, ConstantSizedBufferBuilderOf<int64>, clue, value);

builder.list = builder.list->Prepend(entry, alloc);

if (entryOut)

{

\*entryOut = entry;

}

return sizeof(int64);

}

uint32 PrependByte(BufferBuilderList & builder, LPCWSTR clue, byte value)

{

auto entry = Anew(alloc, BufferBuilderByte, clue, value);

builder.list = builder.list->Prepend(entry, alloc);

return sizeof(byte);

}

uint32 PrependFunctionBodyFlags(BufferBuilderList & builder, LPCWSTR clue, FunctionBody::FunctionBodyFlags value)

{

return PrependByte(builder, clue, (byte) value);

}

uint32 PrependBool(BufferBuilderList & builder, LPCWSTR clue, bool value)

{

return PrependByte(builder, clue, (byte) value);

}

uint32 PrependFloat(BufferBuilderList & builder, LPCWSTR clue, float value)

{

auto entry = Anew(alloc, BufferBuilderFloat, clue, value);

builder.list = builder.list->Prepend(entry, alloc);

return sizeof(float);

}

uint32 PrependDouble(BufferBuilderList & builder, LPCWSTR clue, double value)

{

auto entry = Anew(alloc, BufferBuilderDouble, clue, value);

builder.list = builder.list->Prepend(entry, alloc);

return sizeof(double);

}

uint32 PrependSIMDValue(BufferBuilderList & builder, LPCWSTR clue, SIMDValue value)

{

auto entry = Anew(alloc, BufferBuilderSIMD, clue, value);

builder.list = builder.list->Prepend(entry, alloc);

return sizeof(SIMDValue);

}

uint32 PrependString16(\_\_in BufferBuilderList & builder, \_\_in\_nz LPCWSTR clue, \_\_in\_bcount\_opt(byteLength) LPCWSTR sz, \_\_in uint32 byteLength)

{

if (sz != nullptr)

{

auto bb = Anew(alloc, ByteBuffer, byteLength, (void\*)sz); // Includes trailing null

return PrependInt32(builder, clue, GetString16Id(bb));

}

else

{

return PrependInt32(builder, clue, 0xffffffff);

}

}

uint32 PrependByteBuffer(BufferBuilderList & builder, LPCWSTR clue, ByteBuffer \* bb)

{

auto id = GetString16Id(bb);

return PrependInt32(builder, clue, id);

}

int GetIdOfPropertyRecord(const PropertyRecord \* propertyRecord)

{

AssertMsg(!propertyRecord->IsSymbol(), "bytecode serializer does not currently handle non-built-in symbol PropertyRecords");

size\_t byteCount = ((size\_t)propertyRecord->GetLength() + 1) \* sizeof(wchar\_t);

if (byteCount > UINT\_MAX)

{

// We should never see property record that big

Js::Throw::InternalError();

}

auto buffer = propertyRecord->GetBuffer();

#if DBG

const PropertyRecord \* propertyRecordCheck;

scriptContext->FindPropertyRecord(buffer, propertyRecord->GetLength(), &propertyRecordCheck);

Assert(propertyRecordCheck == propertyRecord);

#endif

auto bb = Anew(alloc, ByteBuffer, (uint32)byteCount, (void\*)buffer);

return GetString16Id(bb, /\*isPropertyRecord=\*/ true);

}

template<typename TLayout>

unaligned TLayout \* DuplicateLayout(unaligned const TLayout \* in)

{

auto sizeOfLayout = sizeof(unaligned TLayout);

auto newLayout = AnewArray(alloc, byte, sizeOfLayout);

js\_memcpy\_s(newLayout, sizeOfLayout, in, sizeOfLayout);

return (unaligned TLayout \* )newLayout;

}

template<typename T>

uint32 Prepend(BufferBuilderList & builder, LPCWSTR clue, T \* t)

{

auto block = Anew(alloc, BufferBuilderRaw, clue, sizeof(serialization\_alignment T), (const byte\*)t);

builder.list = builder.list->Prepend(block, alloc);

return sizeof(serialization\_alignment T);

}

struct AuxRecord

{

SerializedAuxiliaryKind kind;

uint offset;

};

#ifndef TEMP\_DISABLE\_ASMJS

HRESULT RewriteAsmJsByteCodesInto(BufferBuilderList & builder, LPCWSTR clue, FunctionBody \* function, ByteBlock \* byteBlock)

{

SListCounted<AuxRecord> auxRecords(alloc);

Assert(!OmitFunction(function->GetSerializationIndex()));

auto finalSize = Anew(alloc, BufferBuilderInt32, L"Final Byte Code Size", 0); // Initially set to zero

builder.list = builder.list->Prepend(finalSize, alloc);

ByteCodeReader reader;

reader.Create(function);

uint32 size = 0;

const byte \* opStart = nullptr;

bool cantGenerate = false;

auto saveBlock = [&]() {

uint32 byteCount;

if (TryConvertToUInt32(reader.GetIP() - opStart, &byteCount))

{

if (!GenerateByteCodeForNative(function->GetSerializationIndex()))

{

auto block = Anew(alloc, BufferBuilderRaw, clue, byteCount, (const byte\*)opStart);

builder.list = builder.list->Prepend(block, alloc);

size += byteCount;

}

}

else

{

AssertMsg(false, "Unlikely: byte code size overflows 32 bits");

cantGenerate = true;

}

};

Assert(!function->HasCachedScopePropIds());

while (!cantGenerate)

{

opStart = reader.GetIP();

opStart; // For prefast. It can't figure out that opStart is captured in saveBlock above.

LayoutSize layoutSize;

OpCodeAsmJs op = (OpCodeAsmJs)reader.ReadOp(layoutSize);

if (op == OpCodeAsmJs::EndOfBlock)

{

saveBlock();

break;

}

OpLayoutTypeAsmJs layoutType = OpCodeUtilAsmJs::GetOpCodeLayout(op);

switch (layoutType)

{

#define DEFAULT\_LAYOUT(op) \

case OpLayoutTypeAsmJs::##op: { \

Assert(layoutSize == SmallLayout); \

reader.##op(); \

saveBlock(); \

break; }

#define DEFAULT\_LAYOUT\_WITH\_ONEBYTE(op) \

case OpLayoutTypeAsmJs::##op: { \

switch (layoutSize) \

{ \

case SmallLayout: \

reader.##op##\_Small(); \

break; \

case MediumLayout: \

reader.##op##\_Medium(); \

break; \

case LargeLayout: \

reader.##op##\_Large(); \

break; \

default: \

Assume(UNREACHED); \

} \

saveBlock(); \

break; }

DEFAULT\_LAYOUT(Empty);

DEFAULT\_LAYOUT(StartCall);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(ElementSlot);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(AsmTypedArr);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(AsmCall);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(AsmReg1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(AsmReg2);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(AsmReg3);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(AsmReg4);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(AsmReg5);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(AsmReg6);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(AsmReg7);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(AsmReg2IntConst1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int1Double1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int1Float1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Double1Int1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Double1Float1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Double1Reg1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float1Reg1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int1Reg1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Reg1Double1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Reg1Float1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Reg1Int1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int1Double2);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int1Float2);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int2);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int1Const1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int3);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Double2);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float2);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float3);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float1Double1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float1Int1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Double3);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(AsmUnsigned1);

DEFAULT\_LAYOUT(AsmBr);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(BrInt1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(BrInt2);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float32x4\_2);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float32x4\_3);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float32x4\_4);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float32x4\_1Float4);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float32x4\_2Int4);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float32x4\_3Int4);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float32x4\_1Float1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float32x4\_2Float1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float32x4\_1Float64x2\_1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float32x4\_1Int32x4\_1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float32x4\_1Int32x4\_1Float32x4\_2);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Reg1Float32x4\_1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int1Float32x4\_1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int32x4\_2);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int32x4\_3);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int32x4\_4);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int32x4\_1Int1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int32x4\_1Int4);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int32x4\_2Int4);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int32x4\_3Int4);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int32x4\_2Int1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int32x4\_2Int2);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int1Int32x4\_1Int1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float32x4\_2Int1Float1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float1Float32x4\_1Int1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Reg1Int32x4\_1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int32x4\_1Float64x2\_1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int32x4\_1Float32x4\_1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int1Int32x4\_1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float64x2\_2);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float64x2\_3);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float64x2\_4);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float64x2\_1Double2);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float64x2\_1Double1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float64x2\_2Double1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float64x2\_2Int2);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float64x2\_3Int2);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float64x2\_1Float32x4\_1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float64x2\_1Int32x4\_1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Float64x2\_1Int32x4\_1Float64x2\_2);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Reg1Float64x2\_1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Int1Float64x2\_1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(AsmSimdTypedArr);

#undef DEFAULT\_LAYOUT

#undef DEFAULT\_LAYOUT\_WITH\_ONEBYTE

default:

AssertMsg(false, "Unknown OpLayout");

cantGenerate = true;

break;

}

}

if (cantGenerate)

{

return ByteCodeSerializer::CantGenerate;

}

if (size != byteBlock->GetLength() && !GenerateByteCodeForNative(function->GetSerializationIndex()))

{

Assert(size == byteBlock->GetLength());

return ByteCodeSerializer::CantGenerate;

}

finalSize->value = size;

RewriteAuxiliaryInto(builder, auxRecords, reader, function);

return S\_OK;

}

#endif

HRESULT RewriteByteCodesInto(BufferBuilderList & builder, LPCWSTR clue, FunctionBody \* function, ByteBlock \* byteBlock)

{

SListCounted<AuxRecord> auxRecords(alloc);

Assert(!OmitFunction(function->GetSerializationIndex()));

auto finalSize = Anew(alloc, BufferBuilderInt32, L"Final Byte Code Size", 0); // Initially set to zero

builder.list = builder.list->Prepend(finalSize, alloc);

ByteCodeReader reader;

reader.Create(function);

uint32 size = 0;

const byte \* opStart = nullptr;

bool cantGenerate = false;

auto saveBlock = [&]() {

uint32 byteCount;

if (TryConvertToUInt32(reader.GetIP()-opStart, &byteCount))

{

if (!GenerateByteCodeForNative(function->GetSerializationIndex()))

{

auto block = Anew(alloc, BufferBuilderRaw, clue, byteCount, (const byte\*) opStart);

builder.list = builder.list->Prepend(block, alloc);

size += byteCount;

}

}

else

{

AssertMsg(false, "Unlikely: byte code size overflows 32 bits");

cantGenerate = true;

}

};

if (function->HasCachedScopePropIds())

{

AuxRecord record = { sakPropertyIdArrayForCachedScope, 0 };

auxRecords.Prepend(record);

}

while(!cantGenerate)

{

opStart = reader.GetIP();

opStart; // For prefast. It can't figure out that opStart is captured in saveBlock above.

LayoutSize layoutSize;

OpCode op = reader.ReadOp(layoutSize);

if (op == OpCode::EndOfBlock)

{

saveBlock();

break;

}

OpLayoutType layoutType = OpCodeUtil::GetOpCodeLayout(op);

switch (layoutType)

{

#define DEFAULT\_LAYOUT(op) \

case OpLayoutType::##op: { \

Assert(layoutSize == SmallLayout); \

reader.##op(); \

saveBlock(); \

break; }

#define DEFAULT\_LAYOUT\_WITH\_ONEBYTE(op) \

case OpLayoutType::##op: { \

switch (layoutSize) \

{ \

case SmallLayout: \

reader.##op##\_Small(); \

break; \

case MediumLayout: \

reader.##op##\_Medium(); \

break; \

case LargeLayout: \

reader.##op##\_Large(); \

break; \

default: \

Assert(false); \

\_\_assume(false); \

} \

saveBlock(); \

break; }

#define DEFAULT\_LAYOUT\_WITH\_ONEBYTE\_AND\_PROFILED(op) \

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(op); \

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Profiled##op)

DEFAULT\_LAYOUT(Empty);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Reg1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE\_AND\_PROFILED(Reg2);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE\_AND\_PROFILED(Reg2WithICIndex);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE\_AND\_PROFILED(Reg3);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Reg4);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Reg5);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Reg3C);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE\_AND\_PROFILED(Arg);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(ArgNoSrc);

DEFAULT\_LAYOUT(Br);

#ifdef BYTECODE\_BRANCH\_ISLAND

DEFAULT\_LAYOUT(BrLong);

#endif

DEFAULT\_LAYOUT(BrS);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(BrReg1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(BrReg2);

DEFAULT\_LAYOUT(StartCall);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Profiled2CallI);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE\_AND\_PROFILED(CallI);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE\_AND\_PROFILED(CallIFlags);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE\_AND\_PROFILED(CallIWithICIndex);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE\_AND\_PROFILED(CallIFlagsWithICIndex);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE\_AND\_PROFILED(ElementI);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(ElementUnsigned1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE\_AND\_PROFILED(ElementSlot);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE\_AND\_PROFILED(ElementSlotI1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE\_AND\_PROFILED(ElementSlotI2);

DEFAULT\_LAYOUT(W1);

DEFAULT\_LAYOUT(Reg1Int2);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE\_AND\_PROFILED(Reg1Unsigned1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Reg2Int1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Unsigned1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(ElementCP);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(ElementRootCP);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(ElementP);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(ElementPIndexed);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Reg2B1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Reg3B1);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(Class);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(ElementU);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(ElementRootU);

DEFAULT\_LAYOUT(BrProperty);

DEFAULT\_LAYOUT(BrEnvProperty);

DEFAULT\_LAYOUT(BrLocalProperty);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(ElementC2);

DEFAULT\_LAYOUT\_WITH\_ONEBYTE(ElementC);

#undef DEFAULT\_LAYOUT

#undef DEFAULT\_LAYOUT\_WITH\_ONEBYTE

case OpLayoutType::AuxNoReg:

switch (op)

{

case OpCode::InitCachedFuncs:

{

auto layout = reader.AuxNoReg();

AuxRecord record = { sakFuncInfoArray, layout->Offset };

auxRecords.Prepend(record);

saveBlock();

break;

}

case OpCode::CommitScope:

{

// The propertyId array should be saved by the InitCacheScope

auto layout = reader.AuxNoReg();

Assert(layout);

saveBlock();

break;

}

default:

AssertMsg(false, "Unknown OpCode for OpLayoutType::AuxNoReg");

cantGenerate = true;

break;

}

break;

case OpLayoutType::Auxiliary:

switch (op)

{

case OpCode::NewScObjectLiteral:

{

auto layout = reader.Auxiliary();

AuxRecord record = { sakPropertyIdArray, layout->Offset };

auxRecords.Prepend(record);

saveBlock();

break;

}

case OpCode::LdPropIds:

{

auto layout = reader.Auxiliary();

AuxRecord record = { sakPropertyIdArray, layout->Offset };

auxRecords.Prepend(record);

saveBlock();

break;

}

case OpCode::StArrSegItem\_A:

{

auto layout = reader.Auxiliary();

AuxRecord record = { sakVarArrayIntCount, layout->Offset };

auxRecords.Prepend(record);

saveBlock();

break;

}

case OpCode::NewScObject\_A:

{

auto layout = reader.Auxiliary();

AuxRecord record = { sakVarArrayVarCount, layout->Offset };

auxRecords.Prepend(record);

saveBlock();

break;

}

case OpCode::NewScIntArray:

{

auto layout = reader.Auxiliary();

AuxRecord record = { sakIntArray, layout->Offset };

auxRecords.Prepend(record);

saveBlock();

break;

}

case OpCode::NewScFltArray:

{

auto layout = reader.Auxiliary();

AuxRecord record = { sakFloatArray, layout->Offset };

auxRecords.Prepend(record);

saveBlock();

break;

}

default:

AssertMsg(false, "Unknown OpCode for OpLayoutType::Auxiliary");

cantGenerate = true;

break;

}

break;

case OpLayoutType::ProfiledAuxiliary:

switch (op)

{

case OpCode::ProfiledNewScIntArray:

{

auto layout = reader.ProfiledAuxiliary();

AuxRecord record = { sakIntArray, layout->Offset };

auxRecords.Prepend(record);

saveBlock();

break;

}

case OpCode::ProfiledNewScFltArray:

{

auto layout = reader.ProfiledAuxiliary();

AuxRecord record = { sakFloatArray, layout->Offset };

auxRecords.Prepend(record);

saveBlock();

break;

}

default:

AssertMsg(false, "Unknown OpCode for OpLayoutType::ProfiledAuxiliary");

cantGenerate = true;

break;

}

break;

case OpLayoutType::Reg2Aux:

switch (op)

{

case OpCode::SpreadArrayLiteral:

{

auto layout = reader.Reg2Aux();

AuxRecord record = { sakIntArray, layout->Offset };

auxRecords.Prepend(record);

saveBlock();

break;

}

default:

AssertMsg(false, "Unknown OpCode for OpLayoutType::Reg2Aux");

cantGenerate = true;

break;

}

break;

#define STORE\_SPREAD\_AUX\_ARGS \

if (!(layout->Options & CallIExtended\_SpreadArgs)) \

{ \

break; \

} \

AuxRecord record = { sakIntArray, layout->SpreadAuxOffset }; \

auxRecords.Prepend(record)

#define CALLIEXTENDED\_LAYOUT\_WITH\_ONEBYTE(op) \

case OpLayoutType::##op: \

{ \

switch (layoutSize) \

{ \

case SmallLayout: \

{ \

auto layout = reader.##op##\_Small(); \

STORE\_SPREAD\_AUX\_ARGS; \

break; \

} \

case MediumLayout: \

{ \

auto layout = reader.##op##\_Medium(); \

STORE\_SPREAD\_AUX\_ARGS; \

break; \

} \

case LargeLayout: \

{ \

auto layout = reader.##op##\_Large(); \

STORE\_SPREAD\_AUX\_ARGS; \

break; \

} \

default: \

Assert(false); \

\_\_assume(false); \

} \

saveBlock(); \

break; \

}

CALLIEXTENDED\_LAYOUT\_WITH\_ONEBYTE(CallIExtended)

CALLIEXTENDED\_LAYOUT\_WITH\_ONEBYTE(CallIExtendedFlags)

CALLIEXTENDED\_LAYOUT\_WITH\_ONEBYTE(ProfiledCallIExtended)

CALLIEXTENDED\_LAYOUT\_WITH\_ONEBYTE(ProfiledCallIExtendedFlags)

CALLIEXTENDED\_LAYOUT\_WITH\_ONEBYTE(Profiled2CallIExtended)

CALLIEXTENDED\_LAYOUT\_WITH\_ONEBYTE(ProfiledCallIExtendedWithICIndex)

CALLIEXTENDED\_LAYOUT\_WITH\_ONEBYTE(ProfiledCallIExtendedFlagsWithICIndex)

default:

AssertMsg(false, "Unknown OpLayout");

cantGenerate = true;

break;

}

}

if (cantGenerate)

{

return ByteCodeSerializer::CantGenerate;

}

if (size != byteBlock->GetLength() && !GenerateByteCodeForNative(function->GetSerializationIndex()))

{

Assert(size == byteBlock->GetLength());

return ByteCodeSerializer::CantGenerate;

}

finalSize->value = size;

RewriteAuxiliaryInto(builder, auxRecords, reader, function);

return S\_OK;

}

void RewriteAuxiliaryInto(BufferBuilderList& builder, SListCounted<AuxRecord> const& auxRecordList,

ByteCodeReader& reader, FunctionBody \* functionBody)

{

uint count = auxRecordList.Count();

PrependInt32(builder, L"Auxiliary Structure Count", count);

if (count == 0)

{

return;

}

auto writeAuxVarArray = [&](uint offset, bool isVarCount, int count, const Var \* elements) {

typedef serialization\_alignment SerializedVarArray T;

T header(offset, isVarCount, count);

auto block = Anew(alloc, ConstantSizedBufferBuilderOf<T>, L"Var Array", header);

builder.list = builder.list->Prepend(block, alloc);

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

{

auto var = elements[i];

PrependVarConstant(builder, var);

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

PrependInt32(builder, L"Magic end of aux var array", magicEndOfAuxVarArray);

PrependInt32(builder, L"Magic end of aux", magicEndOfAux);

#endif

};

auto writeAuxVarArrayIntCount = [&](uint offset) {

const AuxArray<Var> \* varArray = reader.ReadAuxArray<Var>(offset, functionBody);

int count = varArray->count;

const Var \* elements = varArray->elements;

writeAuxVarArray(offset, false, count, elements);

};

auto writeAuxVarArrayVarCount = [&](uint offset) {

const VarArrayVarCount \* varArray = reader.ReadVarArrayVarCount(offset, functionBody);

int count = Js::TaggedInt::ToInt32(varArray->count);

const Var \* elements = varArray->elements;

writeAuxVarArray(offset, true, count, elements);

};

auto writeAuxIntArray = [&](uint offset) -> BufferBuilder\* {

const AuxArray<int32> \*ints = reader.ReadAuxArray<int32>(offset, functionBody);

int count = ints->count;

const int32 \* elements = ints->elements;

typedef serialization\_alignment SerializedIntArray T;

T header(offset, count);

auto block = Anew(alloc, ConstantSizedBufferBuilderOf<T>, L"Int Array", header);

builder.list = builder.list->Prepend(block, alloc);

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

{

auto value = elements[i];

PrependConstantInt32(builder, L"Integer Constant Value", value);

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

PrependInt32(builder, L"Magic end of aux int array", magicEndOfAuxIntArray);

PrependInt32(builder, L"Magic end of aux", magicEndOfAux);

#endif

return block;

};

auto writeAuxFloatArray = [&](uint offset) -> BufferBuilder\* {

const AuxArray<double> \*doubles = reader.ReadAuxArray<double>(offset, functionBody);

int count = doubles->count;

const double \* elements = doubles->elements;

typedef serialization\_alignment SerializedFloatArray T;

T header(offset, count);

auto block = Anew(alloc, ConstantSizedBufferBuilderOf<T>, L"Float Array", header);

builder.list = builder.list->Prepend(block, alloc);

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

{

auto value = elements[i];

PrependDouble(builder, L"Number Constant Value", value);

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

PrependInt32(builder, L"Magic end of aux float array", magicEndOfAuxFltArray);

PrependInt32(builder, L"Magic end of aux", magicEndOfAux);

#endif

return block;

};

auto writeAuxPropertyIdArray = [&](uint offset, uint32 extraSlots) -> BufferBuilder\* {

const PropertyIdArray \* propIds = reader.ReadPropertyIdArray(offset, functionBody, extraSlots);

typedef serialization\_alignment SerializedPropertyIdArray T;

T header(offset, propIds->count, extraSlots, propIds->hadDuplicates, propIds->has\_\_proto\_\_);

auto block = Anew(alloc, ConstantSizedBufferBuilderOf<T>, L"Property Id Array", header);

builder.list = builder.list->Prepend(block, alloc);

for (uint32 i=0; i<propIds->count; i++)

{

auto original = propIds->elements[i];

auto encoded = encodePossiblyBuiltInPropertyId(original);

PrependConstantInt32(builder, L"Encoded Property Id", encoded);

}

auto slots = propIds->elements + propIds->count;

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

{

PrependConstantInt32(builder, L"Extra Slot", slots[i]);

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

PrependInt32(builder, L"Magic end of aux section", magicEndOfAuxPropIdArray);

PrependInt32(builder, L"Magic end of aux", magicEndOfAux);

#endif

return block;

};

auto writeAuxFuncInfoArray = [&](uint offset) -> BufferBuilder\* {

const FuncInfoArray \* funcInfos = reader.ReadAuxArray<FuncInfoEntry>(offset, functionBody);

typedef serialization\_alignment SerializedFuncInfoArray T;

T header(offset, funcInfos->count);

auto block = Anew(alloc, ConstantSizedBufferBuilderOf<T>, L"Funcinfo Array", header);

builder.list = builder.list->Prepend(block, alloc);

for (uint32 i=0; i<funcInfos->count; i++)

{

auto funcInfo = funcInfos->elements[i];

PrependConstantInt32(builder, L"FuncInfo nestedIndex", funcInfo.nestedIndex);

PrependConstantInt32(builder, L"FuncInfo scopeSlot", funcInfo.scopeSlot);

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

PrependInt32(builder, L"Magic end of aux section", magicEndOfAuxFuncInfoArray);

PrependInt32(builder, L"Magic end of aux", magicEndOfAux);

#endif

return block;

};

PrependInt32(builder, L"Auxiliary Size",

functionBody->GetAuxiliaryData()? functionBody->GetAuxiliaryData()->GetLength() : 0);

PrependInt32(builder, L"Auxiliary Context Size",

functionBody->GetAuxiliaryContextData()? functionBody->GetAuxiliaryContextData()->GetLength() : 0);

auxRecordList.Map([&](AuxRecord const& auxRecord)

{

switch (auxRecord.kind)

{

default:

AssertMsg(false, "Unexpected auxiliary kind");

Throw::FatalInternalError();

break;

case sakVarArrayIntCount:

writeAuxVarArrayIntCount(auxRecord.offset);

break;

case sakVarArrayVarCount:

writeAuxVarArrayVarCount(auxRecord.offset);

break;

case sakIntArray:

writeAuxIntArray(auxRecord.offset);

break;

case sakFloatArray:

writeAuxFloatArray(auxRecord.offset);

break;

case sakPropertyIdArray:

writeAuxPropertyIdArray(auxRecord.offset, 0);

break;

case sakPropertyIdArrayForCachedScope:

writeAuxPropertyIdArray(auxRecord.offset, ActivationObjectEx::ExtraSlotCount());

break;

case sakFuncInfoArray:

writeAuxFuncInfoArray(auxRecord.offset);

break;

};

});

}

uint32 PrependStringConstant(BufferBuilderList & builder, Var var)

{

auto str = JavascriptString::FromVar(var);

uint32 size = 0;

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

size += PrependInt32(builder, L"Start String Constant", magicStartStringConstant);

#endif

auto bb = Anew(alloc, ByteBuffer, (str->GetLength() + 1) \* sizeof(wchar\_t), (void\*)str->GetSz());

size += PrependByteBuffer(builder, L"String Constant 16 Value", bb);

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

size += PrependInt32(builder, L"End String Constant", magicEndStringConstant);

#endif

return size;

}

uint32 PrependStringTemplateCallsiteConstant(BufferBuilderList & builder, Var var)

{

ES5Array\* callsite = ES5Array::FromVar(var);

Var element = nullptr;

auto size = PrependInt32(builder, L"String Template Callsite Constant String Count", (int)callsite->GetLength());

for (uint32 i = 0; i < callsite->GetLength(); i++)

{

callsite->DirectGetItemAt(i, &element);

size += PrependStringConstant(builder, element);

}

Var rawVar = JavascriptOperators::OP\_GetProperty(callsite, Js::PropertyIds::raw, callsite->GetScriptContext());

ES5Array\* rawArray = ES5Array::FromVar(rawVar);

for (uint32 i = 0; i < rawArray->GetLength(); i++)

{

rawArray->DirectGetItemAt(i, &element);

size += PrependStringConstant(builder, element);

}

return size;

}

uint32 PrependVarConstant(BufferBuilderList & builder, Var var)

{

if (var == (Js::Var)&Js::NullFrameDisplay)

{

return PrependByte(builder, L"Null Frame Display", ctNullDisplay);

}

else if (var == (Js::Var)&Js::StrictNullFrameDisplay)

{

return PrependByte(builder, L"Strict Null Frame Display", ctStrictNullDisplay);

}

auto typeId = JavascriptOperators::GetTypeId(var);

switch (typeId)

{

case TypeIds\_Undefined:

return PrependByte(builder, L"Undefined Constant", ctUndefined);

case TypeIds\_Null:

return PrependByte(builder, L"Null Constant", ctNull);

case TypeIds\_Boolean:

return PrependByte(builder, L"Boolean Constant", JavascriptBoolean::FromVar(var)->GetValue()? ctTrue : ctFalse);

case TypeIds\_Number:

{

auto size = PrependByte(builder, L"Number Constant", ctNumber);

return size + PrependDouble(builder, L"Number Constant Value", JavascriptNumber::GetValue(var));

}

case TypeIds\_Integer:

{

auto size = PrependByte(builder, L"Integer Constant", ctInt);

return size + PrependConstantInt32(builder, L"Integer Constant Value", TaggedInt::ToInt32(var));

}

case TypeIds\_String:

{

auto size = PrependByte(builder, L"String Constant 16", ctString16);

return size + PrependStringConstant(builder, var);

}

case TypeIds\_ES5Array:

{

// ES5Array objects in the constant table are always string template callsite objects.

// If we later put other ES5Array objects in the constant table, we'll need another way

// to decide the constant type.

auto size = PrependByte(builder, L"String Template Callsite Constant", ctStringTemplateCallsite);

return size + PrependStringTemplateCallsiteConstant(builder, var);

}

default:

AssertMsg(UNREACHED, "Unexpected object type in AddConstantTable");

Throw::FatalInternalError();

}

}

#ifndef TEMP\_DISABLE\_ASMJS

uint32 AddAsmJsConstantTable(BufferBuilderList & builder, FunctionBody \* function)

{

uint32 size = 0;

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

size += PrependInt32(builder, L"Start Constant Table", magicStartOfConstantTable);

#endif

uint32 intConstCount = function->GetAsmJsFunctionInfo()->GetIntConstCount();

uint32 floatConstCount = function->GetAsmJsFunctionInfo()->GetFloatConstCount();

uint32 doubleConstCount = function->GetAsmJsFunctionInfo()->GetDoubleConstCount();

Js::Var \* constTable = static\_cast<Js::Var \*>(function->GetConstTable());

int \* intConstTable = reinterpret\_cast<int \*>(constTable + Js::AsmJsFunctionMemory::RequiredVarConstants - 1);

for (Js::RegSlot reg = Js::FunctionBody::FirstRegSlot; reg < intConstCount; ++reg)

{

PrependConstantInt32(builder, L"Integer Constant Value", intConstTable[reg]);

}

float \* floatConstTable = reinterpret\_cast<float \*>(intConstTable + intConstCount);

for (Js::RegSlot reg = Js::FunctionBody::FirstRegSlot; reg < floatConstCount; ++reg)

{

PrependFloat(builder, L"Float Constant Value", floatConstTable[reg]);

}

double \* doubleConstTable = reinterpret\_cast<double \*>(floatConstTable + floatConstCount);

for (Js::RegSlot reg = Js::FunctionBody::FirstRegSlot; reg < doubleConstCount; ++reg)

{

PrependDouble(builder, L"Double Constant Value", doubleConstTable[reg]);

}

uint32 simdConstCount = function->GetAsmJsFunctionInfo()->GetSimdConstCount();

AsmJsSIMDValue \*simdConstTable = reinterpret\_cast<AsmJsSIMDValue \*>(doubleConstTable + doubleConstCount);

for (Js::RegSlot reg = Js::FunctionBody::FirstRegSlot; reg < simdConstCount; ++reg)

{

PrependSIMDValue(builder, L"SIMD Constant Value", simdConstTable[reg]);

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

size += PrependInt32(builder, L"End Constant Table", magicEndOfConstantTable);

#endif

return size;

}

#endif

uint32 AddConstantTable(BufferBuilderList & builder, FunctionBody \* function)

{

uint32 size = 0;

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

size += PrependInt32(builder, L"Start Constant Table", magicStartOfConstantTable);

#endif

for (auto reg = FunctionBody::FirstRegSlot + 1; reg < function->m\_constCount; reg++) // Ignore first slot, it is always global object or module root object

{

auto var = function->GetConstantVar(reg);

Assert(var != nullptr);

size += PrependVarConstant(builder, var);

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

size += PrependInt32(builder, L"End Constant Table", magicEndOfConstantTable);

#endif

return size;

}

uint32 AddPropertyIdsForScopeSlotArray(BufferBuilderList & builder, FunctionBody \* function)

{

if (function->scopeSlotArraySize == 0)

{

return 0;

}

uint32 size = 0;

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

size += PrependInt32(builder, L"Start PropertyIdsForScopeSlotsArray", magicStartOfPropertyIdsForScopeSlotArray);

#endif

for (uint i = 0; i < function->scopeSlotArraySize; i++)

{

PropertyId propertyId = encodePossiblyBuiltInPropertyId(function->propertyIdsForScopeSlotArray[i]);

size += PrependInt32(builder, L"PropertyIdsForScopeSlots", propertyId);

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

size += PrependInt32(builder, L"End PropertyIdsForScopeSlotsArray", magicEndOfPropertyIdsForScopeSlotArray);

#endif

return size;

}

// Gets the number of debugger slot array scopes there are in the function body's scope chain list.

uint32 GetDebuggerScopeSlotArrayCount(FunctionBody \* function)

{

Assert(function);

uint debuggerScopeSlotArraySize = 0;

if (function->GetScopeObjectChain())

{

debuggerScopeSlotArraySize = function->GetScopeObjectChain()->pScopeChain->CountWhere([&](DebuggerScope\* scope)

{

return scope->IsSlotScope();

});

}

return debuggerScopeSlotArraySize;

}

// Adds the debugger scopes that are slot array type to the serialized bytecode.

// This is to ensure that block scope slot array properties are captured along with

// function level slot array properties.

uint32 AddSlotArrayDebuggerScopeProperties(BufferBuilderList & builder, DebuggerScope\* debuggerScope, uint propertiesCount)

{

Assert(debuggerScope);

if (propertiesCount == 0)

{

return 0u;

}

uint32 size = 0u;

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

size += PrependInt32(builder, L"Start SlotArrayDebuggerScopeProperties", magicStartOfDebuggerScopeProperties);

#endif // BYTE\_CODE\_MAGIC\_CONSTANTS

AssertMsg(debuggerScope->HasProperties(), "Properties should exist.");

Assert(debuggerScope->scopeProperties->Count() >= 0);

AssertMsg((uint)debuggerScope->scopeProperties->Count() == propertiesCount, "Property counts should match.");

for (uint i = 0u; i < propertiesCount; ++i)

{

DebuggerScopeProperty scopeProperty = debuggerScope->scopeProperties->Item(i);

size += PrependInt32(builder, L"SlotIndexesForDebuggerScopeSlots", scopeProperty.location);

PropertyId propertyId = encodePossiblyBuiltInPropertyId(scopeProperty.propId);

size += PrependInt32(builder, L"PropertyIdsForDebuggerScopeSlots", propertyId);

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

size += PrependInt32(builder, L"End SlotArrayDebuggerScopeProperties", magicEndOfDebuggerScopeProperties);

#endif // BYTE\_CODE\_MAGIC\_CONSTANTS

return size;

}

// Adds the debugger scopes that are slot array type to the serialized bytecode.

// This is to ensure that block scope slot array properties are captured along with

// function level slot array properties.

uint32 AddSlotArrayDebuggerScopes(BufferBuilderList & builder, FunctionBody\* function, uint debuggerScopeSlotArraySize)

{

Assert(function);

if (function->GetScopeObjectChain() == nullptr || debuggerScopeSlotArraySize == 0)

{

return 0u;

}

uint32 size = 0u;

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

size += PrependInt32(builder, L"Start SlotArrayDebuggerScopes", magicStartOfDebuggerScopes);

#endif // BYTE\_CODE\_MAGIC\_CONSTANTS

uint slotArrayCount = 0;

for (uint i = 0u; i < static\_cast<uint>(function->GetScopeObjectChain()->pScopeChain->Count()); ++i)

{

DebuggerScope\* debuggerScope = function->GetScopeObjectChain()->pScopeChain->Item(i);

if (debuggerScope->IsSlotScope())

{

// Only add slot scope debugger scopes (store the index of the scope).

size += PrependInt32(builder, L"SlotArrayDebuggerScope", i);

// Store the count of properties for the scope.

int propertiesCount = debuggerScope->HasProperties() ? debuggerScope->scopeProperties->Count() : 0u;

size += PrependInt32(builder, L"Debugger Scope Slot Array Property Count", propertiesCount);

size += AddSlotArrayDebuggerScopeProperties(builder, debuggerScope, propertiesCount);

slotArrayCount++;

}

}

Assert(debuggerScopeSlotArraySize == slotArrayCount);

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

size += PrependInt32(builder, L"End SlotArrayDebuggerScopes", magicEndOfDebuggerScopes);

#endif // BYTE\_CODE\_MAGIC\_CONSTANTS

return size;

}

uint32 AddCacheIdToPropertyIdMap(BufferBuilderList & builder, FunctionBody \* function)

{

uint count = function->GetInlineCacheCount();

if (count == 0)

{

return 0;

}

uint32 size = 0;

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

PrependInt32(builder, L"Start CacheId-to-PropertyId map", magicStartOfCacheIdToPropIdMap);

#endif

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

{

PropertyId propertyId = encodePossiblyBuiltInPropertyId(function->GetPropertyIdFromCacheId(i));

size += PrependInt32(builder, L"CacheIdToPropertyId", propertyId);

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

size += PrependInt32(builder, L"End CacheId-to-PropertyId map", magicEndOfCacheIdToPropIdMap);

#endif

return size;

}

uint32 AddReferencedPropertyIdMap(BufferBuilderList & builder, FunctionBody \* function)

{

uint count = function->GetReferencedPropertyIdCount();

if (count == 0)

{

return 0;

}

uint32 size = 0;

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

PrependInt32(builder, L"Start Referenced-PropertyId map", magicStartOfReferencedPropIdMap);

#endif

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

{

PropertyId propertyId = encodeNonBuiltinPropertyId(function->GetReferencedPropertyIdWithMapIndex(i));

size += PrependInt32(builder, L"ReferencedPropertyId", propertyId);

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

size += PrependInt32(builder, L"End Referenced-PropertyId map", magicEndOfReferencedPropIdMap);

#endif

return size;

}

uint32 PrependByteArray(BufferBuilderList & builder, int length, byte \* buffer)

{

int size = 0;

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

{

size += PrependByte(builder, L"Byte Array Element", buffer[i]);

}

return size;

}

uint32 PrependUInt32Array(BufferBuilderList & builder, int length, uint32 \* buffer)

{

int size = 0;

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

{

size += PrependConstantInt32(builder, L"UInt32 Array Element", buffer[i]);

}

return size;

}

uint32 PrependGrowingUint32Array(BufferBuilderList & builder, LPCWSTR clue, JsUtil::GrowingUint32HeapArray \* arr)

{

if (arr == nullptr || arr->Count() == 0 || arr->GetLength() == 0 || arr->GetBuffer() == nullptr)

{

return PrependInt32(builder, clue, 0);

}

auto size = PrependInt32(builder, clue, arr->Count());

size += PrependUInt32Array(builder, arr->Count(), arr->GetBuffer());

return size;

}

uint32 PrependSmallSpanSequence(BufferBuilderList & builder, LPCWSTR clue, SmallSpanSequence \* spanSequence)

{

auto size = PrependInt32(builder, clue, spanSequence->baseValue);

size += PrependGrowingUint32Array(builder, L"Statement Buffer", spanSequence->pStatementBuffer);

size += PrependGrowingUint32Array(builder, L"Actual Offset List", spanSequence->pActualOffsetList);

return size;

}

template <typename TStructType>

uint32 PrependStruct(BufferBuilderList & builder, LPWSTR clue, TStructType \* value)

{

auto entry = Anew(alloc, ConstantSizedBufferBuilderOf<TStructType>, clue, \*value);

builder.list = builder.list->Prepend(entry, alloc);

return sizeof(serialization\_alignment TStructType);

}

#ifndef TEMP\_DISABLE\_ASMJS

uint32 AddAsmJsFunctionInfo(BufferBuilderList & builder, FunctionBody \* function)

{

uint32 size = 0;

AsmJsFunctionInfo\* funcInfo = function->GetAsmJsFunctionInfo();

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

PrependInt32(builder, L"Start Asm.js Function Info", magicStartOfAsmJsFuncInfo);

#endif

size += PrependInt32(builder, L"ReturnType", funcInfo->GetReturnType().which());

size += PrependInt32(builder, L"IntConstCount", funcInfo->GetIntConstCount());

size += PrependInt32(builder, L"DoubleConstCount", funcInfo->GetDoubleConstCount());

size += PrependInt32(builder, L"FloatConstCount", funcInfo->GetFloatConstCount());

size += PrependInt16(builder, L"ArgCount", funcInfo->GetArgCount());

size += PrependInt16(builder, L"ArgSize", funcInfo->GetArgByteSize());

size += PrependInt32(builder, L"IntVarCount", funcInfo->GetIntVarCount());

size += PrependInt32(builder, L"DoubleVarCount", funcInfo->GetDoubleVarCount());

size += PrependInt32(builder, L"FloatVarCount", funcInfo->GetFloatVarCount());

size += PrependInt32(builder, L"IntTmpCount", funcInfo->GetIntTmpCount());

size += PrependInt32(builder, L"DoubleTmpCount", funcInfo->GetDoubleTmpCount());

size += PrependInt32(builder, L"FloatTmpCount", funcInfo->GetFloatTmpCount());

size += PrependInt16(builder, L"ArgSizeArrayLength", funcInfo->GetArgSizeArrayLength());

size += PrependUInt32Array(builder, funcInfo->GetArgSizeArrayLength(), funcInfo->GetArgsSizesArray());

size += PrependByteArray(builder, funcInfo->GetArgCount(), (byte\*)funcInfo->GetArgTypeArray());

size += PrependInt32(builder, L"IntByteOffset", funcInfo->GetIntByteOffset());

size += PrependInt32(builder, L"DoubleByteOffset", funcInfo->GetDoubleByteOffset());

size += PrependInt32(builder, L"FloatByteOffset", funcInfo->GetFloatByteOffset());

size += PrependByte(builder, L"IsHeapBufferConst", funcInfo->IsHeapBufferConst());

size += PrependByte(builder, L"UsesHeapBuffer", funcInfo->UsesHeapBuffer());

size += PrependInt32(builder, L"SIMDConstCount", funcInfo->GetSimdConstCount());

size += PrependInt32(builder, L"SIMDVarCount", funcInfo->GetSimdVarCount());

size += PrependInt32(builder, L"SIMDTmpCount", funcInfo->GetSimdTmpCount());

size += PrependInt32(builder, L"SIMDByteOffset", funcInfo->GetSimdByteOffset());

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

size += PrependInt32(builder, L"End Asm.js Function Info", magicEndOfAsmJsFuncInfo);

#endif

return size;

}

uint32 AddAsmJsModuleInfo(BufferBuilderList & builder, FunctionBody \* function)

{

uint32 size = 0;

AsmJsModuleInfo \* moduleInfo = function->GetAsmJsModuleInfo();

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

PrependInt32(builder, L"Start Asm.js Module Info", magicStartOfAsmJsModuleInfo);

#endif

size += PrependInt32(builder, L"ArgInCount", moduleInfo->GetArgInCount());

size += PrependInt32(builder, L"ExportsCount", moduleInfo->GetExportsCount());

size += PrependInt32(builder, L"SlotsCount", moduleInfo->GetSlotsCount());

size += PrependInt32(builder, L"SIMDRegCount", moduleInfo->GetSimdRegCount());

if (moduleInfo->GetExportsCount() > 0)

{

PropertyIdArray \* propArray = moduleInfo->GetExportsIdArray();

size += PrependByte(builder, L"ExportsIdArrayDups", propArray->hadDuplicates);

size += PrependByte(builder, L"ExportsIdArray\_\_proto\_\_", propArray->has\_\_proto\_\_);

size += PrependInt32(builder, L"ExportsIdArrayLength", propArray->count);

for (uint i = 0; i < propArray->count; i++)

{

PropertyId propertyId = encodePossiblyBuiltInPropertyId(propArray->elements[i]);

size += PrependInt32(builder, L"ExportsIdArrayElem", propertyId);

}

size += PrependUInt32Array(builder, moduleInfo->GetExportsCount(), moduleInfo->GetExportsFunctionLocation());

}

size += PrependInt32(builder, L"ExportFunctionIndex", moduleInfo->GetExportFunctionIndex());

size += PrependInt32(builder, L"VarCount", moduleInfo->GetVarCount());

for (int i = 0; i < moduleInfo->GetVarCount(); i++)

{

size += PrependStruct(builder, L"ModuleVar", &moduleInfo->GetVar(i));

}

size += PrependInt32(builder, L"VarImportCount", moduleInfo->GetVarImportCount());

for (int i = 0; i < moduleInfo->GetVarImportCount(); i++)

{

auto import = moduleInfo->GetVarImport(i);

size += PrependInt32(builder, L"ImportLocation", import.location);

size += PrependByte(builder, L"ImportType", import.type);

PropertyId propertyId = encodePossiblyBuiltInPropertyId(import.field);

size += PrependInt32(builder, L"ImportId", propertyId);

}

size += PrependInt32(builder, L"FunctionImportCount", moduleInfo->GetFunctionImportCount());

for (int i = 0; i < moduleInfo->GetFunctionImportCount(); i++)

{

auto import = moduleInfo->GetFunctionImport(i);

size += PrependInt32(builder, L"ImportLocation", import.location);

PropertyId propertyId = encodePossiblyBuiltInPropertyId(import.field);

size += PrependInt32(builder, L"ImportId", propertyId);

}

size += PrependInt32(builder, L"FunctionCount", moduleInfo->GetFunctionCount());

for (int i = 0; i < moduleInfo->GetFunctionCount(); i++)

{

auto func = moduleInfo->GetFunction(i);

size += PrependInt32(builder, L"FuncLocation", func.location);

}

size += PrependInt32(builder, L"FunctionTableCount", moduleInfo->GetFunctionTableCount());

for (int i = 0; i < moduleInfo->GetFunctionTableCount(); i++)

{

auto table = moduleInfo->GetFunctionTable(i);

size += PrependInt32(builder, L"FuncTableSize", table.size);

PrependUInt32Array(builder, table.size, table.moduleFunctionIndex);

}

size += PrependStruct<AsmJsModuleMemory>(builder, L"ModuleMemory", &moduleInfo->GetModuleMemory());

size += PrependInt32(builder, L"AsmJsSlotMapCount", moduleInfo->GetAsmJsSlotMap()->Count());

auto slotIter = moduleInfo->GetAsmJsSlotMap()->GetIterator();

while (slotIter.IsValid())

{

PropertyId propertyId = encodePossiblyBuiltInPropertyId(slotIter.CurrentKey());

size += PrependInt32(builder, L"AsmJsSlotPropId", propertyId);

size += PrependStruct(builder, L"AsmJsSlotValue", slotIter.CurrentValue());

slotIter.MoveNext();

}

size += PrependStruct(builder, L"MathBuiltinBV", &moduleInfo->GetAsmMathBuiltinUsed());

size += PrependStruct(builder, L"ArrayBuiltinBV", &moduleInfo->GetAsmArrayBuiltinUsed());

size += PrependStruct(builder, L"SIMDBuiltinBV", &moduleInfo->GetAsmSimdBuiltinUsed());

size += PrependInt32(builder, L"MaxHeapAccess", moduleInfo->GetMaxHeapAccess());

size += PrependByte(builder, L"UsesChangeHeap", moduleInfo->GetUsesChangeHeap());

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

size += PrependInt32(builder, L"End Asm.js Module Info", magicEndOfAsmJsModuleInfo);

#endif

return size;

}

#endif

HRESULT AddFunctionBody(BufferBuilderList & builder, FunctionBody \* function, SRCINFO const \* srcInfo, int \*serializationIndex)

{

SerializedFieldList definedFields = { 0 };

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

PrependInt32(builder, L"Start Function Table", magicStartOfFunctionBody);

#endif

function->SetSerializationIndex((\*serializationIndex)++);

uint32 sourceDiff = 0;

if (!TryConvertToUInt32(function->StartOffset(), &sourceDiff))

{

Assert(0); // Likely a bug

return ByteCodeSerializer::CantGenerate;

}

PrependString16(builder, L"Display Name", function->m\_displayName, (function->m\_displayNameLength +1)\* sizeof(wchar\_t));

if (function->m\_lineNumber != 0)

{

definedFields.has\_m\_lineNumber = true;

PrependInt32(builder, L"Line Number", function->m\_lineNumber);

}

if (function->m\_columnNumber != 0)

{

definedFields.has\_m\_columnNumber = true;

PrependInt32(builder, L"Column Number", function->m\_columnNumber);

}

// FunctionBody Details

DWORD bitFlags =

(function->m\_isDeclaration ? ffIsDeclaration : 0)

| (function->m\_hasImplicitArgIns ? ffHasImplicitArgsIn : 0)

| (function->m\_isAccessor ? ffIsAccessor : 0)

| (function->m\_isStaticNameFunction ? ffIsStaticNameFunction : 0)

| (function->m\_isNamedFunctionExpression ? ffIsNamedFunctionExpression : 0)

| (function->m\_isNameIdentifierRef ? ffIsNameIdentifierRef : 0)

| (function->m\_isGlobalFunc ? ffIsGlobalFunc : 0)

| (function->m\_dontInline ? ffDontInline : 0)

| (function->m\_isFuncRegistered ? ffIsFuncRegistered : 0)

| (function->m\_isStrictMode ? ffIsStrictMode : 0)

| (function->m\_doBackendArgumentsOptimization ? ffDoBackendArgumentsOptimization : 0)

| (function->m\_isEval ? ffIsEval : 0)

| (function->m\_isDynamicFunction ? ffIsDynamicFunction : 0)

| (function->m\_hasAllNonLocalReferenced ? ffhasAllNonLocalReferenced : 0)

| (function->m\_hasSetIsObject ? ffhasSetIsObject : 0)

| (function->m\_CallsEval ? ffhasSetCallsEval : 0)

| (function->m\_ChildCallsEval ? ffChildCallsEval : 0)

| (function->m\_hasReferenceableBuiltInArguments ? ffHasReferenceableBuiltInArguments : 0)

#ifndef TEMP\_DISABLE\_ASMJS

| (function->m\_isAsmjsMode ? ffIsAsmJsMode : 0)

| (function->m\_isAsmJsFunction ? ffIsAsmJsFunction : 0)

#endif

;

PrependInt32(builder, L"BitFlags", bitFlags);

PrependInt32(builder, L"Relative Function ID", function->functionId - topFunctionId); // Serialized function ids are relative to the top function ID

PrependInt32(builder, L"Serialization ID", function->GetSerializationIndex());

PrependInt32(builder, L"Attributes", function->GetAttributes());

AssertMsg((function->GetAttributes() &

~(FunctionInfo::Attributes::ErrorOnNew

| FunctionInfo::Attributes::SuperReference

| FunctionInfo::Attributes::DefaultConstructor

| FunctionInfo::Attributes::Lambda

| FunctionInfo::Attributes::Async

| FunctionInfo::Attributes::CapturesThis

| FunctionInfo::Attributes::Generator

| FunctionInfo::Attributes::ClassConstructor

| FunctionInfo::Attributes::ClassMethod)) == 0,

"Only the ErrorOnNew|SuperReference|DefaultConstructor|Lambda|CapturesThis|Generator|ClassConstructor|Async|ClassMember attributes should be set on a serialized function");

PrependInt32(builder, L"Offset Into Source", sourceDiff);

if (function->GetNestedCount() > 0)

{

definedFields.has\_m\_nestedCount = true;

PrependInt32(builder, L"Nested count", function->GetNestedCount());

}

// This field should always be non-zero

Assert(function->m\_constCount != 0);

#define PrependArgSlot PrependInt16

#define PrependRegSlot PrependInt32

#define PrependCharCount PrependInt32

#define PrependULong PrependInt32

#define PrependUInt16 PrependInt16

#define PrependUInt32 PrependInt32

#define DEFINE\_FUNCTION\_PROXY\_FIELDS 1

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

#define DECLARE\_SERIALIZABLE\_FIELD(type, name, serializableType) \

if (function->##name != 0) { \

definedFields.has\_##name = true; \

Prepend##serializableType(builder, L#name, function->##name); \

}

#define DECLARE\_MANUAL\_SERIALIZABLE\_FIELD(type, name, serializableType, serializeHere) \

if (function->##name != 0 && serializeHere) { \

definedFields.has\_##name = true; \

Prepend##serializableType(builder, L#name, function->##name); \

}

#include "SerializableFunctionFields.h"

if (!OmitFunction(function->GetSerializationIndex()))

{

#define DEFINE\_FUNCTION\_BODY\_FIELDS 1

#define DECLARE\_SERIALIZABLE\_FIELD(type, name, serializableType) \

if (function->##name != 0) { \

definedFields.has\_##name = true; \

Prepend##serializableType(builder, L#name, function->##name); \

}

#define DECLARE\_MANUAL\_SERIALIZABLE\_FIELD(type, name, serializableType, serializeHere) \

if (function->##name != 0 && serializeHere) { \

definedFields.has\_##name = true; \

Prepend##serializableType(builder, L#name, function->##name); \

}

#include "SerializableFunctionFields.h"

}

if (!OmitFunction(function->GetSerializationIndex()))

{

if (function->loopHeaderArray)

{

PrependByte(builder, L"Loop Header Array Exists", 1);

for (uint i = 0; i < function->loopCount; ++i)

{

PrependInt32(builder, L"Loop Header Start", function->loopHeaderArray[i].startOffset);

PrependInt32(builder, L"Loop Header End", function->loopHeaderArray[i].endOffset);

}

}

else

{

PrependByte(builder, L"Loop Header Array Exists", 0);

}

#ifndef TEMP\_DISABLE\_ASMJS

if (function->GetAsmJsFunctionInfo())

{

PrependByte(builder, L"Asm.js Info Exists", 1);

AddAsmJsFunctionInfo(builder, function);

}

else if (function->GetIsAsmjsMode())

{

PrependByte(builder, L"Asm.js Info Exists", 2);

AddAsmJsModuleInfo(builder, function);

}

else

#endif

{

PrependByte(builder, L"Asm.js Info Exists", 0);

}

#ifndef TEMP\_DISABLE\_ASMJS

if (function->GetIsAsmJsFunction())

{

AddAsmJsConstantTable(builder, function);

auto hr = RewriteAsmJsByteCodesInto(builder, L"Rewritten Byte Code", function, function->byteCodeBlock);

if (FAILED(hr))

{

return hr;

}

}

else

#endif

{

AddConstantTable(builder, function);

auto hr = RewriteByteCodesInto(builder, L"Rewritten Byte Code", function, function->byteCodeBlock);

if (FAILED(hr))

{

return hr;

}

}

AddCacheIdToPropertyIdMap(builder, function);

AddReferencedPropertyIdMap(builder, function);

AddPropertyIdsForScopeSlotArray(builder, function);

uint debuggerScopeSlotArraySize = GetDebuggerScopeSlotArrayCount(function);

PrependInt32(builder, L"Debugger Scope Slot Array Size", debuggerScopeSlotArraySize);

AddSlotArrayDebuggerScopes(builder, function, debuggerScopeSlotArraySize);

// Literal regexes

for (uint i = 0; i < function->literalRegexCount; ++i)

{

const auto literalRegex = function->GetLiteralRegex(i);

if (!literalRegex)

{

PrependInt32(builder, L"Literal regex source length", -1);

continue;

}

const auto source = literalRegex->GetSource();

PrependInt32(builder, L"Literal regex source length", source.GetLength());

PrependString16(builder, L"Literal regex source", source.GetBuffer(), (source.GetLength() + 1)\* sizeof(wchar\_t));

PrependByte(builder, L"Literal regex flags", literalRegex->GetFlags());

}

// Write the SourceInfo stuff

PrependSmallSpanSequence(builder, L"Span Sequence", function->m\_sourceInfo.pSpanSequence);

}

// Lastly, write each of the lexically enclosed functions

if (function->GetNestedCount())

{

auto nestedBodyList = Anew(alloc, BufferBuilderList, L"Nest Function Bodies");

for(uint32 i = 0; i<function->GetNestedCount(); ++i)

{

auto nestedFunctionBody = function->GetNestedFunc(i)->GetFunctionBody();

if (nestedFunctionBody==nullptr)

{

PrependInt32(builder, L"Empty Nested Function", 0);

}

else

{

auto nestedFunctionBuilder = Anew(alloc, BufferBuilderList, L"Nested Function");

nestedBodyList->list = nestedBodyList->list->Prepend(nestedFunctionBuilder, alloc);

auto offsetToNested = Anew(alloc, BufferBuilderRelativeOffset, L"Offset To Nested Function", nestedFunctionBuilder);

builder.list = builder.list->Prepend(offsetToNested, alloc);

AddFunctionBody(\*nestedFunctionBuilder, nestedFunctionBody, srcInfo, serializationIndex);

}

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

PrependInt32(builder, L"End Function Body", magicEndOfFunctionBody);

#endif

builder.list = builder.list->Prepend(nestedBodyList, alloc);

}

else

{

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

PrependInt32(builder, L"End Function Body", magicEndOfFunctionBody);

#endif

}

// Increment the function count

++functionCount.value;

// Reverse to put prepended items in correct order

builder.list = builder.list->ReverseCurrentList();

PrependStruct<SerializedFieldList>(builder, L"Serialized Field List", &definedFields);

return S\_OK;

}

HRESULT AddTopFunctionBody(FunctionBody \* function, SRCINFO const \* srcInfo)

{

topFunctionId = function->functionId;

int serializationIndex = 0;

return AddFunctionBody(functionsTable, function, srcInfo, &serializationIndex);

}

};

class ByteCodeBufferReader

{

public:

ScriptContext \* scriptContext;

byte \* raw;

int magic;

int totalSize;

byte fileVersionScheme;

int V1;

int V2;

int V3;

int V4;

byte architecture;

int expectedFunctionBodySize;

int expectedBuildInPropertyCount;

int expectedOpCodeCount;

int firstFunctionId;

int functionCount;

const byte \* string16s;

int string16Count;

const unaligned StringIndexRecord \* string16IndexTable;

const byte \* string16Table;

const byte \* sourceSpans;

int lineInfoCacheCount;

const byte \* lineInfoCaches;

const JsUtil::LineOffsetCache<Recycler>::LineOffsetCacheItem \* lineInfoCache;

const byte \* functions;

int sourceSize;

int sourceCharLength;

Utf8SourceInfo \*utf8SourceInfo;

uint sourceIndex;

bool const isLibraryCode;

public:

ByteCodeBufferReader(ScriptContext \* scriptContext, byte \* raw, bool isLibraryCode, int builtInPropertyCount)

: scriptContext(scriptContext), raw(raw), utf8SourceInfo(nullptr), isLibraryCode(isLibraryCode),

expectedFunctionBodySize(sizeof(unaligned FunctionBody)),

expectedBuildInPropertyCount(builtInPropertyCount),

expectedOpCodeCount((int)OpCode::Count)

{

if (isLibraryCode)

{

expectedFunctionBodySize = 0;

expectedOpCodeCount = 0;

}

}

static const byte\* ReadFunctionBodyFlags(const byte \* buffer, size\_t remainingBytes, FunctionBody::FunctionBodyFlags \* value)

{

Assert(remainingBytes >= sizeof(FunctionBody::FunctionBodyFlags));

\*value = \*(FunctionBody::FunctionBodyFlags\*) buffer;

return buffer + sizeof(FunctionBody::FunctionBodyFlags);

}

const byte\* ReadFunctionBodyFlags(const byte \* buffer, FunctionBody::FunctionBodyFlags \* value)

{

auto remainingBytes = (raw + totalSize) - buffer;

return ReadFunctionBodyFlags(buffer, remainingBytes, value);

}

const byte\* ReadBool(const byte \* buffer, bool \* value)

{

auto remainingBytes = (raw + totalSize) - buffer;

Assert(remainingBytes >= sizeof(bool));

\*value = \*buffer ? true : false;

return buffer + sizeof(bool);

}

static const byte \* ReadByte(const byte \* buffer, size\_t remainingBytes, byte \* value)

{

Assert(remainingBytes>=sizeof(byte));

\*value = \*(byte\*)buffer;

return buffer + sizeof(byte);

}

const byte \* ReadByte(const byte \* buffer, byte \* value)

{

auto remainingBytes = (raw + totalSize) - buffer;

return ReadByte(buffer, remainingBytes, value);

}

static const byte \* ReadInt16(const byte \* buffer, size\_t remainingBytes, int16 \* value)

{

#if VARIABLE\_INT\_ENCODING

return ReadVariableInt<int16>(buffer, remainingBytes, value);

#else

Assert(remainingBytes>=sizeof(int16));

\*value = \*(int16 \*) buffer;

return buffer + sizeof(int16);

#endif

}

const byte \* ReadInt16(const byte \* buffer, int16 \* value)

{

auto remainingBytes = (raw + totalSize) - buffer;

return ReadInt16(buffer, remainingBytes, value);

}

static const byte \* ReadConstantSizedInt64(const byte \* buffer, size\_t remainingBytes, int64 \* value)

{

Assert(remainingBytes >= sizeof(int64));

\*value = \*(int64 \*)buffer;

return buffer + sizeof(int64);

}

const byte \* ReadConstantSizedInt64(const byte \* buffer, int64 \* value)

{

auto remainingBytes = (raw + totalSize) - buffer;

return ReadConstantSizedInt64(buffer, remainingBytes, value);

}

static const byte \* ReadConstantSizedInt32(const byte \* buffer, size\_t remainingBytes, int \* value)

{

Assert(remainingBytes >= sizeof(int));

\*value = \*(int \*) buffer;

return buffer + sizeof(int);

}

const byte \* ReadConstantSizedInt32(const byte \* buffer, int \* value)

{

auto remainingBytes = (raw + totalSize) - buffer;

return ReadConstantSizedInt32(buffer, remainingBytes, value);

}

static const byte \* ReadInt32(const byte \* buffer, size\_t remainingBytes, int \* value)

{

#if VARIABLE\_INT\_ENCODING

return ReadVariableInt<int>(buffer, remainingBytes, value);

#else

Assert(remainingBytes >= sizeof(int));

return ReadConstantSizedInt32(buffer, remainingBytes, value);

#endif

}

const byte \* ReadInt32(const byte \* buffer, int \* value)

{

auto remainingBytes = (raw + totalSize) - buffer;

return ReadInt32(buffer, remainingBytes, value);

}

const byte \* ReadCharCount(const byte \* buffer, size\_t remainingBytes, charcount\_t \* value)

{

Assert(remainingBytes>=sizeof(charcount\_t));

#if VARIABLE\_INT\_ENCODING

return ReadVariableInt<charcount\_t>(buffer, remainingBytes, value);

#else

\*value = \*(charcount\_t \*) buffer;

return buffer + sizeof(charcount\_t);

#endif

}

const byte \* ReadCharCount(const byte \* buffer, charcount\_t \* value)

{

auto remainingBytes = (raw + totalSize) - buffer;

return ReadCharCount(buffer, remainingBytes, value);

}

static const byte \* ReadFloat(const byte \* buffer, size\_t remainingBytes, float \* value)

{

Assert(remainingBytes >= sizeof(float));

\*value = \*(float \*)buffer;

return buffer + sizeof(float);

}

const byte \* ReadFloat(const byte \* buffer, float \* value)

{

auto remainingBytes = (raw + totalSize) - buffer;

return ReadFloat(buffer, remainingBytes, value);

}

static const byte \* ReadDouble(const byte \* buffer, size\_t remainingBytes, double \* value)

{

Assert(remainingBytes>=sizeof(double));

\*value = \*(double \*)buffer;

return buffer + sizeof(double);

}

const byte \* ReadDouble(const byte \* buffer, double \* value)

{

auto remainingBytes = (raw + totalSize) - buffer;

return ReadDouble(buffer, remainingBytes, value);

}

static const byte \* ReadSIMDValue(const byte \* buffer, size\_t remainingBytes, SIMDValue \* value)

{

Assert(remainingBytes >= sizeof(SIMDValue));

\*value = \*(SIMDValue \*)buffer;

return buffer + sizeof(SIMDValue);

}

const byte \* ReadSIMDValue(const byte \* buffer, SIMDValue \* value)

{

auto remainingBytes = (raw + totalSize) - buffer;

return ReadSIMDValue(buffer, remainingBytes, value);

}

const byte \* ReadUInt16(const byte \* buffer, uint16 \* value)

{

auto remainingBytes = (raw + totalSize) - buffer;

return ReadInt16(buffer, remainingBytes, (int16\*)value);

}

const byte \* ReadUInt32(const byte \* buffer, unsigned int \* value)

{

auto remainingBytes = (raw + totalSize) - buffer;

return ReadInt32(buffer, remainingBytes, (int\*)value);

}

const byte \* ReadULong(const byte \* buffer, ulong \* value)

{

auto remainingBytes = (raw + totalSize) - buffer;

return ReadInt32(buffer, remainingBytes, (int\*)value);

}

const byte \* ReadRegSlot(const byte \* buffer, RegSlot \* value)

{

auto remainingBytes = (raw + totalSize) - buffer;

return ReadInt32(buffer, remainingBytes, (int\*)value);

}

const byte \* ReadArgSlot(const byte \* buffer, ArgSlot \* value)

{

auto remainingBytes = (raw + totalSize) - buffer;

return ReadInt32(buffer, remainingBytes, (int\*)value);

}

const byte \* ReadConstantSizedInt32NoSize(const byte \* buffer, int \* value)

{

\*value = \*(int \*)buffer;

return buffer + sizeof(int);

}

const byte \* ReadOffsetAsPointer(const byte \* buffer, byte const \*\* value)

{

int offset;

auto next = ReadConstantSizedInt32(buffer, &offset);

if (offset == 0)

{

\*value = nullptr;

return next;

}

\*value = raw + offset;

return next;

}

const byte \* ReadByteBlock(const byte \* buffer, WriteBarrierPtr<ByteBlock>\* byteBlock)

{

int contentLength;

buffer = ReadInt32(buffer, &contentLength);

if (contentLength == 0)

{

\*byteBlock = nullptr;

}

else

{

// TODO: Abstract this out to ByteBlock::New

\*byteBlock = RecyclerNewLeaf(scriptContext->GetRecycler(), ByteBlock, contentLength, (byte\*)buffer);

}

return buffer + contentLength;

}

const byte \* ReadAuxiliary(const byte \* buffer, FunctionBody \* functionBody)

{

const byte \* current = buffer;

uint32 countOfAuxiliaryStructure;

current = ReadUInt32(current, &countOfAuxiliaryStructure);

if (countOfAuxiliaryStructure == 0)

{

return current;

}

uint32 sizeOfAuxiliaryBlock;

uint32 sizeOfAuxiliaryContextBlock;

current = ReadUInt32(current, &sizeOfAuxiliaryBlock);

current = ReadUInt32(current, &sizeOfAuxiliaryContextBlock);

ByteBlock \* auxBlock = sizeOfAuxiliaryBlock?

ByteBlock::New(scriptContext->GetRecycler(), nullptr, sizeOfAuxiliaryBlock) : nullptr;

ByteBlock \* auxContextBlock = sizeOfAuxiliaryContextBlock?

ByteBlock::New(scriptContext->GetRecycler(), nullptr, sizeOfAuxiliaryContextBlock) : nullptr;

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

{

typedef serialization\_alignment const SerializedAuxiliary TBase;

auto part = (serialization\_alignment const SerializedAuxiliary \* )current;

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

Assert(part->auxMagic == magicStartOfAux);

#endif

switch(part->kind)

{

default:

AssertMsg(false, "Unexpected auxiliary kind");

Throw::FatalInternalError();

break;

case sakVarArrayIntCount:

current = DeserializeVarArray<VarArray>(scriptContext, current, auxBlock);

break;

case sakVarArrayVarCount:

current = DeserializeVarArray<VarArrayVarCount>(scriptContext, current, auxContextBlock);

break;

case sakIntArray:

current = DeserializeIntArray(scriptContext, current, auxBlock);

break;

case sakFloatArray:

current = DeserializeFloatArray(scriptContext, current, auxBlock);

break;

case sakPropertyIdArray:

current = DeserializePropertyIdArray(scriptContext, current, auxBlock, functionBody);

break;

case sakFuncInfoArray:

current = DeserializeFuncInfoArray(scriptContext, current, auxBlock);

break;

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int magicEnd;

current = ReadInt32(current, &magicEnd);

Assert(magicEnd == magicEndOfAux);

#endif

}

functionBody->auxBlock = auxBlock;

functionBody->auxContextBlock = auxContextBlock;

return current;

}

LPCWSTR GetString16ById(int id, bool\* isPropertyRecord = nullptr)

{

if (id == 0xffffffff)

{

return nullptr;

}

if(!(id >= this->expectedBuildInPropertyCount && id <= string16Count + this->expectedBuildInPropertyCount))

{

Assert(false);

}

const unaligned StringIndexRecord\* record = string16IndexTable + (id - this->expectedBuildInPropertyCount);

if(isPropertyRecord)

{

\*isPropertyRecord = record->isProprertyRecord;

}

auto offset = record->offset;

auto addressOfString = raw + offset;

return (LPCWSTR)addressOfString;

}

uint32 GetString16LengthById(int id)

{

if(!(id >= this->expectedBuildInPropertyCount && id<=string16Count + this->expectedBuildInPropertyCount))

{

Assert(false);

}

LPCWSTR s1 = GetString16ById(id);

LPCWSTR s2 = GetString16ById(id + 1);

auto result = s2 - s1 - 1;

Assert(result <= UINT\_MAX);

return (uint32)result;

}

HRESULT ReadHeader()

{

auto current = ReadConstantSizedInt32NoSize(raw, &magic);

if (magic != magicConstant)

{

AssertMsg(false, "Unrecognized magic constant in byte code file header. Is this really a bytecode file?");

return E\_FAIL;

}

current = ReadConstantSizedInt32NoSize(current, &totalSize);

current = ReadByte(current, &fileVersionScheme);

byte expectedFileVersionScheme = isLibraryCode? ReleaseVersioningScheme : CurrentFileVersionScheme;

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

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

{

expectedFileVersionScheme = (byte)Js::Configuration::Global.flags.ForceSerializedBytecodeVersionSchema;

}

#endif

// Ignore the version scheme check if it is library code

if (!isLibraryCode && fileVersionScheme != expectedFileVersionScheme)

{

// File version scheme is incompatible.

return ByteCodeSerializer::InvalidByteCode;

}

DWORD expectedV1 = 0;

DWORD expectedV2 = 0;

DWORD expectedV3 = 0;

DWORD expectedV4 = 0;

if (expectedFileVersionScheme != ReleaseVersioningScheme)

{

Js::VerifyCatastrophic(!isLibraryCode);

Js::VerifyCatastrophic(expectedFileVersionScheme == EngineeringVersioningScheme);

Js::VerifyOkCatastrophic(AutoSystemInfo::GetJscriptFileVersion(&expectedV1, &expectedV2, &expectedV3, &expectedV4));

}

else

{

auto guidDWORDs = (DWORD\*)(&byteCodeCacheReleaseFileVersion);

expectedV1 = guidDWORDs[0];

expectedV2 = guidDWORDs[1];

expectedV3 = guidDWORDs[2];

expectedV4 = guidDWORDs[3];

}

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

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

{

expectedV1 = Js::Configuration::Global.flags.ForceSerializedBytecodeMajorVersion;

expectedV2 = 0;

expectedV3 = 0;

expectedV4 = 0;

}

#endif

current = ReadConstantSizedInt32(current, &V1);

if ((DWORD)V1!=expectedV1)

{

// Incompatible major version

return ByteCodeSerializer::InvalidByteCode;

}

// Library code is serialized with one build of the engine and then included into a subsequent build, so can't have match

// on the build timestamp hash. Also want to share the generated bytecode between x86/ARM and debug/release, so skip the extra

// checking. Will rework this validation entirely under TFS 555060

current = ReadConstantSizedInt32(current, &V2);

if ((DWORD)V2 != expectedV2)

{

// Incompatible minor version

return ByteCodeSerializer::InvalidByteCode;

}

current = ReadConstantSizedInt32(current, &V3);

if ((DWORD)V3 != expectedV3)

{

// Incompatible 3rd version part

return ByteCodeSerializer::InvalidByteCode;

}

current = ReadConstantSizedInt32(current, &V4);

if ((DWORD)V4 != expectedV4)

{

// Incompatible 4th version part

return ByteCodeSerializer::InvalidByteCode;

}

current = ReadByte(current, &architecture);

if (architecture!=magicArchitecture)

{

// This byte cache file was created with against a chakra running under a different architecture. It is incompatible.

return ByteCodeSerializer::InvalidByteCode;

}

int functionBodySize, buildInPropertyCount, opCodeCount;

current = ReadInt32(current, &functionBodySize);

if (functionBodySize != expectedFunctionBodySize)

{

// The size of function body didn't match. It is incompatible.

return ByteCodeSerializer::InvalidByteCode;

}

current = ReadInt32(current, &buildInPropertyCount);

if (buildInPropertyCount!=expectedBuildInPropertyCount)

{

// This byte cache file was created with against a chakra that has a different number of built in properties. It is incompatible.

return ByteCodeSerializer::InvalidByteCode;

}

current = ReadInt32(current, &opCodeCount);

if (opCodeCount != expectedOpCodeCount)

{

// This byte cache file was created with against a chakra that has a different number of built in properties. It is incompatible.

return ByteCodeSerializer::InvalidByteCode;

}

current = ReadInt32(current, &sourceSize);

current = ReadInt32(current, &sourceCharLength);

current = ReadOffsetAsPointer(current, &string16s);

current = ReadOffsetAsPointer(current, &lineInfoCaches);

current = ReadOffsetAsPointer(current, &sourceSpans);

current = ReadOffsetAsPointer(current, &functions);

// Read strings header

string16IndexTable = (StringIndexRecord\*)ReadInt32(string16s, &string16Count);

lineInfoCache = (JsUtil::LineOffsetCache<Recycler>::LineOffsetCacheItem \*)ReadInt32(lineInfoCaches, &lineInfoCacheCount);

string16Table = (byte\*)(string16IndexTable + string16Count + 1);

// string16Table is aligned to 2-bytes

uint32 string16TableOffset = (uint32)(string16Table - raw);

string16TableOffset = ::Math::Align(string16TableOffset, (uint32)sizeof(wchar\_t));

string16Table = raw + string16TableOffset;

return S\_OK;

}

const byte\* ReadStringConstant(const byte\* current, FunctionBody\* function, LPCWSTR& string, uint32& len)

{

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int constant;

current = ReadInt32(current, &constant);

Assert(constant == magicStartStringConstant);

#endif

int stringId;

current = ReadInt32(current, &stringId);

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

current = ReadInt32(current, &constant);

Assert(constant == magicEndStringConstant);

#endif

string = GetString16ById(stringId);

len = GetString16LengthById(stringId);

return current;

}

const byte\* ReadStringTemplateCallsiteConstant(const byte\* current, FunctionBody\* function, Var& var)

{

int arrayLength = 0;

current = ReadInt32(current, &arrayLength);

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

ENTER\_PINNED\_SCOPE(Js::JavascriptArray, callsite);

callsite = scriptContext->GetLibrary()->CreateArray(arrayLength);

LPCWSTR string;

uint32 len;

uint32 rawlen = 0;

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

{

current = ReadStringConstant(current, function, string, len);

JavascriptString\* str = JavascriptString::NewCopyBuffer(string, len, scriptContext);

callsite->SetItemWithAttributes(i, str, PropertyEnumerable);

}

JavascriptArray\* rawArray = scriptContext->GetLibrary()->CreateArray(arrayLength);

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

{

current = ReadStringConstant(current, function, string, len);

rawlen += len;

JavascriptString\* str = JavascriptString::NewCopyBuffer(string, len, scriptContext);

rawArray->SetItemWithAttributes(i, str, PropertyEnumerable);

}

rawArray->Freeze();

callsite->SetPropertyWithAttributes(Js::PropertyIds::raw, rawArray, PropertyNone, nullptr);

callsite->Freeze();

JavascriptLibrary\* library = scriptContext->GetLibrary();

var = library->TryGetStringTemplateCallsiteObject(callsite);

if (var == nullptr)

{

library->AddStringTemplateCallsiteObject(callsite);

var = callsite;

}

LEAVE\_PINNED\_SCOPE();

return current;

}

#ifndef TEMP\_DISABLE\_ASMJS

const byte \* ReadAsmJsConstantsTable(const byte \* current, FunctionBody \* function)

{

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int constant;

current = ReadInt32(current, &constant);

Assert(constant == magicStartOfConstantTable);

#endif

function->CreateConstantTable();

uint32 intConstCount = function->GetAsmJsFunctionInfo()->GetIntConstCount();

uint32 floatConstCount = function->GetAsmJsFunctionInfo()->GetFloatConstCount();

uint32 doubleConstCount = function->GetAsmJsFunctionInfo()->GetDoubleConstCount();

Js::Var \* constTable = static\_cast<Js::Var \*>(function->GetConstTable());

int \* intConstTable = reinterpret\_cast<int \*>(constTable + Js::AsmJsFunctionMemory::RequiredVarConstants - 1);

for (Js::RegSlot reg = Js::FunctionBody::FirstRegSlot; reg < intConstCount; ++reg)

{

current = ReadConstantSizedInt32(current, &intConstTable[reg]);

}

float \* floatConstTable = reinterpret\_cast<float \*>(intConstTable + intConstCount);

for (Js::RegSlot reg = Js::FunctionBody::FirstRegSlot; reg < floatConstCount; ++reg)

{

current = ReadFloat(current, &floatConstTable[reg]);

}

double \* doubleConstTable = reinterpret\_cast<double \*>(floatConstTable + floatConstCount);

for (Js::RegSlot reg = Js::FunctionBody::FirstRegSlot; reg < doubleConstCount; ++reg)

{

current = ReadDouble(current, &doubleConstTable[reg]);

}

uint32 simdConstCount = function->GetAsmJsFunctionInfo()->GetSimdConstCount();

AsmJsSIMDValue \*simdConstTable = reinterpret\_cast<AsmJsSIMDValue \*>(doubleConstTable + doubleConstCount);

for (Js::RegSlot reg = Js::FunctionBody::FirstRegSlot; reg < simdConstCount; ++reg)

{

current = ReadSIMDValue(current, &simdConstTable[reg]);

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

current = ReadInt32(current, &constant);

Assert(constant == magicEndOfConstantTable);

#endif

return current;

}

#endif

const byte \* ReadConstantsTable(const byte \* current, FunctionBody \* function)

{

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int constant;

current = ReadInt32(current, &constant);

Assert(constant == magicStartOfConstantTable);

#endif

function->CreateConstantTable();

for (auto reg = FunctionBody::FirstRegSlot + 1; reg < function->m\_constCount; reg++) // Ignore first slot, it is always global or module root and has been preinitialized.

{

byte ct;

current = ReadByte(current, &ct);

switch(ct)

{

case ctString16:

{

LPCWSTR string;

uint32 len;

current = ReadStringConstant(current, function, string, len);

function->RecordStrConstant(reg, string, len);

break;

}

case ctStringTemplateCallsite:

{

Var callsite;

current = ReadStringTemplateCallsiteConstant(current, function, callsite);

function->RecordConstant(reg, callsite);

break;

}

case ctInt:

{

int value;

current = ReadConstantSizedInt32(current, &value);

function->RecordIntConstant(reg, value);

break;

}

case ctNull:

function->RecordNullObject(reg);

break;

case ctUndefined:

function->RecordUndefinedObject(reg);

break;

case ctNumber:

{

double value;

current = ReadDouble(current, &value);

function->RecordFloatConstant(reg, value);

break;

}

case ctNullDisplay:

function->RecordNullDisplayConstant(reg);

break;

case ctStrictNullDisplay:

function->RecordStrictNullDisplayConstant(reg);

break;

case ctTrue:

function->RecordTrueObject(reg);

break;

case ctFalse:

function->RecordFalseObject(reg);

break;

default:

AssertMsg(UNREACHED, "Unexpected object type in ReadConstantsTable");

break;

}

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

current = ReadInt32(current, &constant);

Assert(constant == magicEndOfConstantTable);

#endif

return current;

}

const byte \* ReadPropertyIdsForScopeSlotArray(const byte \* current, FunctionBody \* function)

{

if (function->scopeSlotArraySize == 0)

{

return current;

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int constant;

current = ReadInt32(current, &constant);

Assert(constant == magicStartOfPropertyIdsForScopeSlotArray);

#endif

function->propertyIdsForScopeSlotArray = RecyclerNewArrayLeaf(scriptContext->GetRecycler(), Js::PropertyId, function->scopeSlotArraySize);

for (uint i = 0; i < function->scopeSlotArraySize; i++)

{

int value;

current = ReadInt32(current, &value);

PropertyId propertyId = function->GetByteCodeCache()->LookupPropertyId(value);

function->propertyIdsForScopeSlotArray[i] = propertyId;

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

current = ReadInt32(current, &constant);

Assert(constant == magicEndOfPropertyIdsForScopeSlotArray);

#endif

return current;

}

const byte \* ReadSlotArrayDebuggerScopeProperties(const byte \* current, FunctionBody\* function, DebuggerScope\* debuggerScope, uint propertyCount)

{

Assert(function);

Assert(debuggerScope);

if (propertyCount == 0)

{

return current;

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int constant;

current = ReadInt32(current, &constant);

Assert(constant == magicStartOfDebuggerScopeProperties);

#endif // BYTE\_CODE\_MAGIC\_CONSTANTS

for (uint i = 0u; i < propertyCount; ++i)

{

// Read the slot array index and property ID for each property (for heap enum to use). The remaining properties

// are needed for the debugger and will be filled in when generating byte code.

int value;

current = ReadInt32(current, &value);

RegSlot slotIndex = value;

current = ReadInt32(current, &value);

PropertyId propertyId = function->GetByteCodeCache()->LookupPropertyId(value);

debuggerScope->AddProperty(slotIndex, propertyId, DebuggerScopePropertyFlags\_None);

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

current = ReadInt32(current, &constant);

Assert(constant == magicEndOfDebuggerScopeProperties);

#endif // BYTE\_CODE\_MAGIC\_CONSTANTS

return current;

}

const byte \* ReadSlotArrayDebuggerScopes(const byte \* current, FunctionBody \* function, uint debuggerScopeCount)

{

Assert(function);

if (debuggerScopeCount == 0)

{

return current;

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int constant;

current = ReadInt32(current, &constant);

Assert(constant == magicStartOfDebuggerScopes);

#endif // BYTE\_CODE\_MAGIC\_CONSTANTS

AssertMsg(function->GetScopeObjectChain() == nullptr, "Scope chain should not exist before deserialization.");

function->SetScopeObjectChain(RecyclerNew(scriptContext->GetRecycler(), ScopeObjectChain, scriptContext->GetRecycler()));

int currentScopeOffset = 0;

for (uint i = 0u; i < debuggerScopeCount; i++)

{

int scopeIndex;

current = ReadInt32(current, &scopeIndex);

DebuggerScope\* slotArrayDebuggerScope = nullptr;

AssertMsg(currentScopeOffset <= scopeIndex, "Scope indices were not inserted into the serialized byte code in ascending order.");

while (currentScopeOffset <= scopeIndex)

{

// Fill the chain with dummy scopes until we reach the slot array scope we're on.

// These non-slot array scopes are only needed for the debugger and will be filled in

// properly during byte code generation (when attaching).

// We also don't need to worry about the parenting/sibling chain, as this will be built when

// generating bytecode as well.

slotArrayDebuggerScope = function->AddScopeObject(Js::DiagUnknownScope, 0, Constants::NoRegister);

++currentScopeOffset;

}

Assert(slotArrayDebuggerScope);

// Create the slot array properties.

int propertyCount;

current = ReadInt32(current, &propertyCount);

current = ReadSlotArrayDebuggerScopeProperties(current, function, slotArrayDebuggerScope, propertyCount);

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

current = ReadInt32(current, &constant);

Assert(constant == magicEndOfDebuggerScopes);

#endif // BYTE\_CODE\_MAGIC\_CONSTANTS

return current;

}

const byte \* ReadCacheIdToPropertyIdMap(const byte \* current, FunctionBody \* function)

{

uint count = function->GetInlineCacheCount();

if (count == 0)

{

return current;

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int constant;

current = ReadInt32(current, &constant);

Assert(constant == magicStartOfCacheIdToPropIdMap);

#endif

function->CreateCacheIdToPropertyIdMap();

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

{

int value;

current = ReadInt32(current, &value);

PropertyId propertyId = function->GetByteCodeCache()->LookupPropertyId(value);

function->SetPropertyIdForCacheId(i, propertyId);

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

current = ReadInt32(current, &constant);

Assert(constant == magicEndOfCacheIdToPropIdMap);

#endif

#if DBG

function->VerifyCacheIdToPropertyIdMap();

#endif

return current;

}

const byte \* ReadReferencedPropertyIdMap(const byte \* current, FunctionBody \* function)

{

uint count = function->GetReferencedPropertyIdCount();

if (count == 0)

{

return current;

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int constant;

current = ReadInt32(current, &constant);

Assert(constant == magicStartOfReferencedPropIdMap);

#endif

function->CreateReferencedPropertyIdMap();

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

{

int value;

current = ReadInt32(current, &value);

PropertyId propertyId = function->GetByteCodeCache()->LookupNonBuiltinPropertyId(value);

function->SetReferencedPropertyIdWithMapIndex(i, propertyId);

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

current = ReadInt32(current, &constant);

Assert(constant == magicEndOfReferencedPropIdMap);

#endif

#if DBG

function->VerifyReferencedPropertyIdMap();

#endif

return current;

}

// Read a growing Uint32 array

const byte \* ReadGrowingUint32Array(const byte \* current, JsUtil::GrowingUint32HeapArray \*\* arr)

{

int count = 0;

current = ReadInt32(current, &count);

if (count == 0)

{

(\*arr) = nullptr;

return current;

}

(\*arr) = JsUtil::GrowingUint32HeapArray::Create(/\*length=\*/count);

js\_memcpy\_s((\*arr)->GetBuffer(), count \* sizeof(uint32), current, count\*sizeof(uint32));

(\*arr)->SetCount(count);

current += count \* sizeof(uint32);

return current;

}

// Read a small span sequence

const byte \* ReadSmallSpanSequence(const byte \* current, SmallSpanSequence \*\* smallSpanSequence)

{

(\*smallSpanSequence) = HeapNew(SmallSpanSequence);

current = ReadInt32(current, &(\*smallSpanSequence)->baseValue);

current = ReadGrowingUint32Array(current, &(\*smallSpanSequence)->pStatementBuffer); // CONSIDER: It would be really nice to change GrowingUint32Array to something with a fixed, readonly layout

current = ReadGrowingUint32Array(current, &(\*smallSpanSequence)->pActualOffsetList);

return current;

}

void ReadSourceInfo(const byte \* functionBytes, int& lineNumber, int& columnNumber, bool& m\_isEval, bool& m\_isDynamicFunction)

{

int displayNameId;

unsigned int bitflags;

this->ReadFunctionBodyHeader(functionBytes, displayNameId, lineNumber, columnNumber, bitflags);

m\_isEval = (bitflags & ffIsEval) ? true : false;

m\_isDynamicFunction = (bitflags & ffIsDynamicFunction) ? true : false;

}

const byte \* ReadFunctionBodyHeader(const byte \* functionBytes, int& displayNameId, int& lineNumber, int& columnNumber, unsigned int& bitflags)

{

serialization\_alignment SerializedFieldList\* definedFields = (serialization\_alignment SerializedFieldList\*) functionBytes;

// Basic function body constructor arguments

const byte \* current = functionBytes + sizeof(serialization\_alignment SerializedFieldList);

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int constant;

current = ReadInt32(current, &constant);

Assert(constant == magicStartOfFunctionBody);

#endif

current = ReadInt32(current, &displayNameId);

if (definedFields->has\_m\_lineNumber)

{

current = ReadInt32(current, &lineNumber);

}

else

{

lineNumber = 0;

}

if (definedFields->has\_m\_columnNumber)

{

current = ReadInt32(current, &columnNumber);

}

else

{

columnNumber = 0;

}

current = ReadUInt32(current, &bitflags);

return current;

}

#ifndef TEMP\_DISABLE\_ASMJS

const byte \* ReadAsmJsFunctionInfo(const byte \* current, FunctionBody \* function)

{

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int constant;

current = ReadInt32(current, &constant);

Assert(constant == magicStartOfAsmJsFuncInfo);

#endif

AsmJsFunctionInfo\* funcInfo = function->AllocateAsmJsFunctionInfo();

int retVal;

current = ReadInt32(current, &retVal);

funcInfo->SetReturnType(AsmJsRetType((AsmJsRetType::Which)retVal));

int count;

current = ReadInt32(current, &count);

funcInfo->SetIntConstCount(count);

current = ReadInt32(current, &count);

funcInfo->SetDoubleConstCount(count);

current = ReadInt32(current, &count);

funcInfo->SetFloatConstCount(count);

ArgSlot argCount;

current = ReadUInt16(current, &argCount);

funcInfo->SetArgCount(argCount);

ArgSlot argByteSize;

current = ReadUInt16(current, &argByteSize);

funcInfo->SetArgByteSize(argByteSize);

current = ReadInt32(current, &count);

funcInfo->SetIntVarCount(count);

current = ReadInt32(current, &count);

funcInfo->SetDoubleVarCount(count);

current = ReadInt32(current, &count);

funcInfo->SetFloatVarCount(count);

current = ReadInt32(current, &count);

funcInfo->SetIntTmpCount(count);

current = ReadInt32(current, &count);

funcInfo->SetDoubleTmpCount(count);

current = ReadInt32(current, &count);

funcInfo->SetFloatTmpCount(count);

ArgSlot argSizeArrayLength;

current = ReadUInt16(current, &argSizeArrayLength);

funcInfo->SetArgSizeArrayLength(argSizeArrayLength);

uint\* argArray = RecyclerNewArrayLeafZ(scriptContext->GetRecycler(), uint, argSizeArrayLength);

funcInfo->SetArgsSizesArray(argArray);

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

{

int32 size;

current = ReadConstantSizedInt32(current, &size);

argArray[i] = (uint32)size;

}

if (argCount > 0)

{

AsmJsVarType::Which \* typeArray = RecyclerNewArrayLeaf(scriptContext->GetRecycler(), AsmJsVarType::Which, argCount);

funcInfo->SetArgTypeArray(typeArray);

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

{

current = ReadByte(current, (byte\*)&typeArray[i]);

}

}

current = ReadInt32(current, &count);

funcInfo->SetIntByteOffset(count);

current = ReadInt32(current, &count);

funcInfo->SetDoubleByteOffset(count);

current = ReadInt32(current, &count);

funcInfo->SetFloatByteOffset(count);

bool boolVal;

current = ReadBool(current, &boolVal);

funcInfo->SetIsHeapBufferConst(boolVal);

current = ReadBool(current, &boolVal);

funcInfo->SetUsesHeapBuffer(boolVal);

current = ReadInt32(current, &count);

funcInfo->SetSimdConstCount(count);

current = ReadInt32(current, &count);

funcInfo->SetSimdVarCount(count);

current = ReadInt32(current, &count);

funcInfo->SetSimdTmpCount(count);

current = ReadInt32(current, &count);

funcInfo->SetSimdByteOffset(count);

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

current = ReadInt32(current, &constant);

Assert(constant == magicEndOfAsmJsFuncInfo);

#endif

return current;

}

const byte \* ReadAsmJsModuleInfo(const byte \* current, FunctionBody \* function)

{

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int constant;

current = ReadInt32(current, &constant);

Assert(constant == magicStartOfAsmJsModuleInfo);

#endif

AsmJsModuleInfo\* moduleInfo = function->AllocateAsmJsModuleInfo();

int count;

current = ReadInt32(current, &count);

moduleInfo->SetArgInCount(count);

int exportsCount;

current = ReadInt32(current, &exportsCount);

moduleInfo->SetExportsCount(exportsCount);

current = ReadInt32(current, &count);

moduleInfo->InitializeSlotMap(count);

current = ReadInt32(current, &count);

moduleInfo->SetSimdRegCount(count);

int id;

if (exportsCount > 0)

{

PropertyIdArray \* propArray = moduleInfo->GetExportsIdArray();

bool boolVal;

current = ReadBool(current, &boolVal);

propArray->hadDuplicates = boolVal;

current = ReadBool(current, &boolVal);

propArray->has\_\_proto\_\_ = boolVal;

current = ReadInt32(current, &count);

propArray->count = count;

for (uint i = 0; i < propArray->count; i++)

{

current = ReadInt32(current, &id);

PropertyId propertyId = function->GetByteCodeCache()->LookupPropertyId(id);

propArray->elements[i] = propertyId;

}

RegSlot\* exportLocations = moduleInfo->GetExportsFunctionLocation();

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

{

int32 loc;

current = ReadConstantSizedInt32(current, &loc);

exportLocations[i] = (uint32)loc;

}

}

RegSlot regSlot;

current = ReadUInt32(current, &regSlot);

moduleInfo->SetExportFunctionIndex(regSlot);

current = ReadInt32(current, &count);

moduleInfo->SetVarCount(count);

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

{

serialization\_alignment AsmJsModuleInfo::ModuleVar \* modVar = (serialization\_alignment AsmJsModuleInfo::ModuleVar\*)current;

moduleInfo->SetVar(i, \*modVar);

current = current + sizeof(serialization\_alignment AsmJsModuleInfo::ModuleVar);

}

current = ReadInt32(current, &count);

moduleInfo->SetVarImportCount(count);

AsmJsModuleInfo::ModuleVarImport varImport;

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

{

current = ReadUInt32(current, &varImport.location);

current = ReadByte(current, (byte\*)&varImport.type);

current = ReadInt32(current, &id);

varImport.field = function->GetByteCodeCache()->LookupPropertyId(id);

moduleInfo->SetVarImport(i, varImport);

}

current = ReadInt32(current, &count);

moduleInfo->SetFunctionImportCount(count);

AsmJsModuleInfo::ModuleFunctionImport funcImport;

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

{

current = ReadUInt32(current, &funcImport.location);

current = ReadInt32(current, &id);

funcImport.field = function->GetByteCodeCache()->LookupPropertyId(id);

moduleInfo->SetFunctionImport(i, funcImport);

}

current = ReadInt32(current, &count);

moduleInfo->SetFunctionCount(count);

AsmJsModuleInfo::ModuleFunction modFunc;

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

{

current = ReadUInt32(current, &modFunc.location);

moduleInfo->SetFunction(i, modFunc);

}

current = ReadInt32(current, &count);

moduleInfo->SetFunctionTableCount(count);

AsmJsModuleInfo::ModuleFunctionTable funcTable;

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

{

current = ReadUInt32(current, &funcTable.size);

if (funcTable.size > 0)

{

funcTable.moduleFunctionIndex = RecyclerNewArray(this->scriptContext->GetRecycler(), RegSlot, funcTable.size);

}

else

{

funcTable.moduleFunctionIndex = nullptr;

}

for (uint j = 0; j < funcTable.size; j++)

{

current = ReadConstantSizedInt32(current, (int32\*)&funcTable.moduleFunctionIndex[j]);

}

moduleInfo->SetFunctionTable(i, funcTable);

}

serialization\_alignment AsmJsModuleMemory \* modMem = (serialization\_alignment AsmJsModuleMemory\*)current;

moduleInfo->SetModuleMemory(\*modMem);

current = current + sizeof(serialization\_alignment AsmJsModuleMemory);

current = ReadInt32(current, &count);

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

{

current = ReadInt32(current, &id);

PropertyId key = function->GetByteCodeCache()->LookupPropertyId(id);

serialization\_alignment AsmJsSlot \* slot = (serialization\_alignment AsmJsSlot\*)current;

current = current + sizeof(serialization\_alignment AsmJsSlot);

// copy the slot to recycler memory

AsmJsSlot \* recyclerSlot = RecyclerNew(scriptContext->GetRecycler(), AsmJsSlot);

\*recyclerSlot = \*slot;

moduleInfo->GetAsmJsSlotMap()->Add(key, recyclerSlot);

}

serialization\_alignment BVStatic<ASMMATH\_BUILTIN\_SIZE> \* mathBV = (serialization\_alignment BVStatic<ASMMATH\_BUILTIN\_SIZE>\*)current;

current = current + sizeof(serialization\_alignment BVStatic<ASMMATH\_BUILTIN\_SIZE>);

moduleInfo->SetAsmMathBuiltinUsed(\*mathBV);

serialization\_alignment BVStatic<ASMARRAY\_BUILTIN\_SIZE> \* arrayBV = (serialization\_alignment BVStatic<ASMARRAY\_BUILTIN\_SIZE>\*)current;

current = current + sizeof(serialization\_alignment BVStatic<ASMARRAY\_BUILTIN\_SIZE>);

moduleInfo->SetAsmArrayBuiltinUsed(\*arrayBV);

serialization\_alignment BVStatic<ASMSIMD\_BUILTIN\_SIZE> \* simdBV = (serialization\_alignment BVStatic<ASMSIMD\_BUILTIN\_SIZE>\*)current;

current = current + sizeof(serialization\_alignment BVStatic<ASMSIMD\_BUILTIN\_SIZE>);

moduleInfo->SetAsmSimdBuiltinUsed(\*simdBV);

uint maxAccess;

current = ReadUInt32(current, &maxAccess);

moduleInfo->SetMaxHeapAccess(maxAccess);

bool usesChangeHeap;

current = ReadBool(current, &usesChangeHeap);

moduleInfo->SetUsesChangeHeap(usesChangeHeap);

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

current = ReadInt32(current, &constant);

Assert(constant == magicEndOfAsmJsModuleInfo);

#endif

return current;

}

#endif

// Read a function body

HRESULT ReadFunctionBody(const byte \* functionBytes, FunctionProxy \*\* functionProxy, Utf8SourceInfo\* sourceInfo, ByteCodeCache \* cache, NativeModule \*nativeModule, bool deserializeThis, bool deserializeNested = true, Js::DeferDeserializeFunctionInfo\* deferDeserializeFunctionInfo = NULL)

{

Assert(sourceInfo->GetSrcInfo()->moduleID == kmodGlobal);

int displayNameId;

int lineNumber;

int columnNumber;

unsigned int bitflags;

const byte \* current = this->ReadFunctionBodyHeader(functionBytes, displayNameId, lineNumber, columnNumber, bitflags);

serialization\_alignment SerializedFieldList\* definedFields = (serialization\_alignment SerializedFieldList\*) functionBytes;

auto displayName = deferDeserializeFunctionInfo != nullptr ?

deferDeserializeFunctionInfo->GetDisplayName() :

GetString16ById(displayNameId);

uint displayNameLength = deferDeserializeFunctionInfo ? deferDeserializeFunctionInfo->GetDisplayNameLength() : GetString16LengthById(displayNameId);

uint displayShortNameOffset = deferDeserializeFunctionInfo ? deferDeserializeFunctionInfo->GetShortDisplayNameOffset() : 0;

int functionId;

current = ReadInt32(current, &functionId);

int serializationIndex;

current = ReadInt32(current, &serializationIndex);

int32 attributes;

current = ReadInt32(current, &attributes);

uint32 offsetIntoSource = 0;

current = ReadUInt32(current, &offsetIntoSource);

int nestedCount = 0;

if (definedFields->has\_m\_nestedCount)

{

current = ReadInt32(current, &nestedCount);

}

if (!deserializeThis)

{

Assert(sourceInfo->GetSrcInfo()->moduleID == kmodGlobal);

Assert(!deserializeNested);

\*functionProxy = DeferDeserializeFunctionInfo::New(this->scriptContext, nestedCount, functionId, cache, functionBytes, sourceInfo, displayName, displayNameLength, displayShortNameOffset, nativeModule, (FunctionInfo::Attributes)attributes);

return S\_OK;

}

ParseableFunctionInfo \*\*function = (ParseableFunctionInfo \*\*) functionProxy;

uint functionNumber;

if (deferDeserializeFunctionInfo)

{

functionNumber = deferDeserializeFunctionInfo->GetFunctionNumber();

}

else

{

functionNumber = scriptContext->GetThreadContext()->NewFunctionNumber();

}

if (definedFields->has\_m\_constCount)

{

FunctionBody \*\*functionBody = (FunctionBody \*\*) function;

\*functionBody = FunctionBody::NewFromRecycler(this->scriptContext, nullptr /\*displayName\*/, 0 /\*displayNameLength\*/, 0 /\*displayShortNameOffset\*/, nestedCount,

sourceInfo,

functionNumber,

sourceInfo->GetSrcInfo()->sourceContextInfo->sourceContextId,

firstFunctionId + functionId, nullptr, (FunctionInfo::Attributes)attributes

#ifdef PERF\_COUNTERS

, (deferDeserializeFunctionInfo != nullptr)

#endif

);

(\*functionBody)->SetDisplayName(displayName, displayNameLength, displayShortNameOffset, FunctionProxy::SetDisplayNameFlags::SetDisplayNameFlagsDontCopy);

(\*functionBody)->serializationIndex = serializationIndex;

(\*functionBody)->byteCodeCache = cache;

(\*functionBody)->m\_utf8SourceInfo = utf8SourceInfo; // Set source info

(\*function)->m\_utf8SourceHasBeenSet = true;

}

else

{

\*function = ParseableFunctionInfo::New(this->scriptContext, nestedCount, firstFunctionId + functionId, utf8SourceInfo, displayName, displayNameLength, displayShortNameOffset, nullptr, (FunctionInfo::Attributes)attributes);

}

// These fields are manually deserialized previously

(\*function)->m\_lineNumber = lineNumber;

(\*function)->m\_columnNumber = columnNumber;

(\*function)->m\_isDeclaration = (bitflags & ffIsDeclaration) ? true : false;

(\*function)->m\_hasImplicitArgIns = (bitflags & ffHasImplicitArgsIn) ? true : false;

(\*function)->m\_isAccessor = (bitflags & ffIsAccessor) ? true : false;

(\*function)->m\_isStaticNameFunction = (bitflags & ffIsStaticNameFunction) ? true : false;

(\*function)->m\_isNamedFunctionExpression = (bitflags & ffIsNamedFunctionExpression) ? true : false;

(\*function)->m\_isNameIdentifierRef = (bitflags & ffIsNameIdentifierRef ) ? true : false;

(\*function)->m\_isGlobalFunc = (bitflags & ffIsGlobalFunc) ? true : false;

(\*function)->m\_dontInline = (bitflags & ffDontInline) ? true : false;

(\*function)->m\_isStrictMode = (bitflags & ffIsStrictMode) ? true : false;

(\*function)->m\_doBackendArgumentsOptimization = (bitflags & ffDoBackendArgumentsOptimization) ? true : false;

(\*function)->m\_isEval = (bitflags & ffIsEval) ? true : false;

(\*function)->m\_isDynamicFunction = (bitflags & ffIsDynamicFunction) ? true : false;

// This is offsetIntoSource is the start offset in bytes as well.

(\*function)->m\_cbStartOffset = (size\_t) offsetIntoSource;

(\*function)->m\_sourceIndex = this->sourceIndex;

#define DEFINE\_FUNCTION\_PROXY\_FIELDS 1

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

#define DECLARE\_SERIALIZABLE\_FIELD(type, name, serializableType) \

if (definedFields->has\_##name == true) { \

current = Read##serializableType(current, &(\*function)->##name); \

}

#define DECLARE\_MANUAL\_SERIALIZABLE\_FIELD(type, name, serializableType, deserializeHere) \

if (deserializeHere && definedFields->has\_##name == true) { \

current = Read##serializableType(current, &(\*function)->##name); \

}

#include "SerializableFunctionFields.h"

if (definedFields->has\_m\_constCount)

{

FunctionBody \*\*functionBody = (FunctionBody \*\*)function;

#define DEFINE\_FUNCTION\_BODY\_FIELDS 1

#define DECLARE\_SERIALIZABLE\_FIELD(type, name, serializableType) \

if (definedFields->has\_##name == true) { \

current = Read##serializableType(current, &(\*functionBody)->##name); \

}

#define DECLARE\_MANUAL\_SERIALIZABLE\_FIELD(type, name, serializableType, deserializeHere) \

if (deserializeHere && definedFields->has\_##name == true) { \

current = Read##serializableType(current, &(\*functionBody)->##name); \

}

#include "SerializableFunctionFields.h"

// TODO-STACK-NESTED-FUNC: Defer deserialize function doesn't have parent pointer, can't do stack nested func yet

// The flags field is set to by default to Flags\_HasNoExplicitReturnValue which means if it's serialized, the field will be set

// in the definedFields struct. If it's not set, that means that the flag was explicitly set to Flags\_None so we'll have to set

// that here.

if (definedFields->has\_flags == false)

{

(\*functionBody)->flags = FunctionBody::FunctionBodyFlags::Flags\_None;

}

else

{

(\*functionBody)->flags = (FunctionBody::FunctionBodyFlags)((\*functionBody)->flags & ~FunctionBody::Flags\_StackNestedFunc);

}

if (definedFields->has\_m\_firstTmpReg == false)

{

(\*functionBody)->m\_firstTmpReg = 0;

}

if (definedFields->has\_m\_envDepth == false)

{

(\*functionBody)->m\_envDepth = 0;

}

if (deserializeThis && !deserializeNested)

{

(\*functionBody)->m\_isPartialDeserializedFunction = true;

}

(\*functionBody)->FinishSourceInfo(); // SourceInfo is complete. Register this functionBody to utf8SourceInfo.

(\*functionBody)->m\_isFuncRegistered = (bitflags & ffIsFuncRegistered) ? true : false;

(\*functionBody)->m\_hasAllNonLocalReferenced = (bitflags & ffhasAllNonLocalReferenced) ? true : false;

(\*functionBody)->m\_hasSetIsObject = (bitflags & ffhasSetIsObject) ? true : false;

(\*functionBody)->m\_CallsEval = (bitflags & ffhasSetCallsEval) ? true : false;

(\*functionBody)->m\_ChildCallsEval = (bitflags & ffChildCallsEval) ? true : false;

(\*functionBody)->m\_hasReferenceableBuiltInArguments = (bitflags & ffHasReferenceableBuiltInArguments) ? true : false;

#ifndef TEMP\_DISABLE\_ASMJS

(\*functionBody)->m\_isAsmJsFunction = (bitflags & ffIsAsmJsFunction) ? true : false;

(\*functionBody)->m\_isAsmjsMode = (bitflags & ffIsAsmJsMode) ? true : false;

#endif

byte loopHeaderExists;

current = ReadByte(current, &loopHeaderExists);

if (loopHeaderExists)

{

(\*functionBody)->AllocateLoopHeaders();

for (uint i = 0; i < (\*functionBody)->loopCount; ++i)

{

uint startOffset, endOffset;

current = ReadUInt32(current, &startOffset);

current = ReadUInt32(current, &endOffset);

(\*functionBody)->loopHeaderArray[i].startOffset = startOffset;

(\*functionBody)->loopHeaderArray[i].endOffset = endOffset;

}

}

byte asmJsInfoExists;

current = ReadByte(current, &asmJsInfoExists);

#ifndef TEMP\_DISABLE\_ASMJS

if (asmJsInfoExists == 1)

{

current = ReadAsmJsFunctionInfo(current, \*functionBody);

}

else if (asmJsInfoExists == 2)

{

current = ReadAsmJsModuleInfo(current, \*functionBody);

}

else

#endif

{

Assert(asmJsInfoExists == 0);

}

// Read constants table

#ifndef TEMP\_DISABLE\_ASMJS

if ((\*functionBody)->GetIsAsmJsFunction())

{

current = ReadAsmJsConstantsTable(current, \*functionBody);

}

else

#endif

{

current = ReadConstantsTable(current, \*functionBody);

}

// Byte code

current = ReadByteBlock(current, &(\*functionBody)->byteCodeBlock);

// Auxiliary

current = ReadAuxiliary(current, \*functionBody);

// Inline cache

current = ReadCacheIdToPropertyIdMap(current, \*functionBody);

current = ReadReferencedPropertyIdMap(current, \*functionBody);

(\*functionBody)->AllocateInlineCache();

current = ReadPropertyIdsForScopeSlotArray(current, \*functionBody);

uint debuggerScopeCount = 0;

current = ReadUInt32(current, &debuggerScopeCount);

current = ReadSlotArrayDebuggerScopes(current, \*functionBody, debuggerScopeCount);

(\*functionBody)->AllocateObjectLiteralTypeArray();

// Literal regexes

(\*functionBody)->AllocateLiteralRegexArray();

for (uint i = 0; i < (\*functionBody)->literalRegexCount; ++i)

{

int length;

current = ReadInt32(current, &length);

if (length == -1)

{

Assert(!(\*functionBody)->GetLiteralRegex(i));

continue;

}

int sourceId;

current = ReadInt32(current, &sourceId);

const auto source = GetString16ById(sourceId);

UnifiedRegex::RegexFlags flags;

CompileAssert(sizeof(flags) == sizeof(byte));

current = ReadByte(current, reinterpret\_cast<byte \*>(&flags));

(\*functionBody)->SetLiteralRegex(i, RegexHelper::CompileDynamic(scriptContext, source, length, flags, true));

}

// Read source information

current = ReadSmallSpanSequence(current, &(\*functionBody)->m\_sourceInfo.pSpanSequence);

(\*functionBody)->InitializeExecutionModeAndLimits();

}

// Read lexically nested functions

if (nestedCount)

{

for(auto i = 0; i<nestedCount; ++i)

{

const byte \* nestedFunctionBytes;

current = ReadOffsetAsPointer(current, &nestedFunctionBytes);

if (nestedFunctionBytes == nullptr)

{

(\*function)->SetNestedFunc(NULL, i, 0u);

}

else

{

FunctionProxy\* nestedFunction;

// If we should deserialize nested functions, go ahead and do so

// If we shouldn't, and we're currently deserializing a function proxy

// that has been defer-deserialized, simply copy over the function proxy from

// from the old function- otherwise, create proxies for the nested functions

auto hr = ReadFunctionBody(nestedFunctionBytes, &nestedFunction, sourceInfo, cache, nativeModule, deserializeNested, deserializeNested);

if (FAILED(hr))

{

Assert(0);

return hr;

}

(\*function)->SetNestedFunc(nestedFunction, i, 0u);

}

}

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

// Magical ending

int constant;

ReadInt32(current, &constant);

if (constant != magicEndOfFunctionBody)

{

Assert(constant == magicEndOfFunctionBody);

Throw::FatalInternalError();

}

#endif

if (definedFields->has\_m\_constCount)

{

FunctionBody \*\*functionBody = (FunctionBody \*\*) function;

#if DBG

if (PHASE\_DUMP(Js::DebuggerScopePhase, (\*functionBody)))

{

(\*functionBody)->DumpScopes();

}

#endif

#if ENABLE\_NATIVE\_CODEGEN

if ((!PHASE\_OFF(Js::BackEndPhase, \*functionBody))

&& !this->scriptContext->GetConfig()->IsNoNative()

&& !(\*functionBody)->GetIsAsmjsMode())

{

GenerateFunction(this->scriptContext->GetNativeCodeGenerator(), \*functionBody);

}

#endif // ENABLE\_NATIVE\_CODEGEN

(\*functionBody)->m\_isPartialDeserializedFunction = false;

}

else

{

\*function = (\*function)->Parse(nullptr, true);

}

return S\_OK;

}

// Read the top function body.

HRESULT ReadTopFunctionBody(FunctionBody\*\* function, Utf8SourceInfo\* sourceInfo, ByteCodeCache \* cache, bool allowDefer, NativeModule \*nativeModule)

{

auto topFunction = ReadInt32(functions, &functionCount);

firstFunctionId = sourceInfo->GetSrcInfo()->sourceContextInfo->nextLocalFunctionId;

sourceInfo->GetSrcInfo()->sourceContextInfo->nextLocalFunctionId += functionCount;

sourceInfo->EnsureInitialized(functionCount);

sourceInfo->GetSrcInfo()->sourceContextInfo->EnsureInitialized();

#if ENABLE\_NATIVE\_CODEGEN && defined(ENABLE\_PREJIT)

bool prejit = false;

prejit = (!scriptContext->GetConfig()->IsNoNative() && Js::Configuration::Global.flags.Prejit && nativeModule == nullptr);

allowDefer = allowDefer && !prejit;

#endif

FunctionBody\* functionBody = NULL;

auto result = ReadFunctionBody(topFunction, (FunctionProxy \*\*)&functionBody, sourceInfo, cache, nativeModule, true, !allowDefer /\* don't deserialize nested if defer is allowed \*/);

(\*function) = functionBody;

#if ENABLE\_NATIVE\_CODEGEN && defined(ENABLE\_PREJIT)

if (prejit)

{

Assert(!allowDefer);

GenerateAllFunctions(scriptContext->GetNativeCodeGenerator(), functionBody);

}

#endif

return result;

}

// Deserialize and save a PropertyIdArray

const byte \*

DeserializePropertyIdArray(ScriptContext \* scriptContext, const byte \* buffer, ByteBlock \* deserializeInto, FunctionBody \* functionBody)

{

auto serialized = (serialization\_alignment const Js::SerializedPropertyIdArray \*)buffer;

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

Assert(serialized->magic == magicStartOfAuxPropIdArray);

#endif

auto propertyCount = serialized->propertyCount;

auto extraSlotCount = serialized->extraSlots;

Assert(serialized->offset + sizeof(PropertyIdArray) < deserializeInto->GetLength());

auto result = (PropertyIdArray \*)(deserializeInto->GetBuffer() + serialized->offset);

result->count = propertyCount;

Assert(serialized->offset + result->GetDataSize(extraSlotCount) <= deserializeInto->GetLength());

result->hadDuplicates = serialized->hadDuplicates;

result->has\_\_proto\_\_ = serialized->has\_\_proto\_\_;

auto elements = (PropertyId\*)(serialized + 1);

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

{

result->elements[i] = functionBody->GetByteCodeCache()->LookupPropertyId(elements[i]);

}

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

{

result->elements[propertyCount + i] = elements[propertyCount + i];

}

auto current = buffer +

sizeof(serialization\_alignment const Js::SerializedPropertyIdArray) + (propertyCount + extraSlotCount) \* sizeof(PropertyId);

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int magicEnd;

current = ReadInt32(current, &magicEnd);

Assert(magicEnd == magicEndOfAuxPropIdArray);

#endif

return current;

}

// Deserialize and save a FuncInfoArray

const byte \*

DeserializeFuncInfoArray(ScriptContext \* scriptContext, const byte \* buffer, ByteBlock \* deserializeInto)

{

auto serialized = (serialization\_alignment const SerializedFuncInfoArray \*)buffer;

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

Assert(serialized->magic == magicStartOfAuxFuncInfoArray);

#endif

auto count = serialized->count;

Assert(serialized->offset + sizeof(AuxArray<FuncInfoEntry>) < deserializeInto->GetLength());

auto result = (AuxArray<FuncInfoEntry> \*)(deserializeInto->GetBuffer() + serialized->offset);

result->count = count;

Assert(serialized->offset + result->GetDataSize() <= deserializeInto->GetLength());

auto elements = (int\*)(serialized+1);

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

{

result->elements[i].nestedIndex = elements[i\*2];

result->elements[i].scopeSlot = elements[i\*2+1];

}

auto current = buffer + sizeof(serialization\_alignment const SerializedFuncInfoArray) + (count \* 2 \* sizeof(int));

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int magicEnd;

current = ReadInt32(current, &magicEnd);

Assert(magicEnd == magicEndOfAuxFuncInfoArray);

#endif

return current;

}

// Deserialize a var array

template<typename T>

const byte \* DeserializeVarArray(ScriptContext \* scriptContext, const byte \* buffer, ByteBlock \* deserializeInto)

{

auto serialized = (serialization\_alignment const Js::SerializedVarArray \*)buffer;

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

Assert(serialized->magic == magicStartOfAuxVarArray);

#endif

Assert(serialized->offset + sizeof(T) < deserializeInto->GetLength());

auto result = (T \*)(deserializeInto->GetBuffer() + serialized->offset);

uint count = serialized->varCount;

result->SetCount(count);

Assert(serialized->offset + result->GetDataSize() <= deserializeInto->GetLength());

auto content = (const byte\*)(serialized + 1);

auto current = content;

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

{

byte code;

current = ReadByte(current, &code);

switch(code)

{

case ctInt:

{

int value;

current = ReadConstantSizedInt32(current, &value);

result->elements[index] = Js::TaggedInt::ToVarUnchecked(value);

break;

}

case ctNumber:

{

double value;

current = ReadDouble(current, &value);

const auto number = Js::JavascriptNumber::New(value, scriptContext);

#if !FLOATVAR

scriptContext->BindReference(number);

#endif

result->elements[index] = number;

break;

}

default:

AssertMsg(UNREACHED, "Unexpected object type in VarArray");

Throw::FatalInternalError();

}

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int magicEnd;

current = ReadInt32(current, &magicEnd);

Assert(magicEnd == magicEndOfAuxVarArray);

#endif

return current;

}

const byte \* DeserializeIntArray(ScriptContext \* scriptContext, const byte \* buffer, ByteBlock \* deserializeInto)

{

auto serialized = (serialization\_alignment const Js::SerializedIntArray \*)buffer;

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

Assert(serialized->magic == magicStartOfAuxIntArray);

#endif

Assert(serialized->offset + sizeof(AuxArray<int>) < deserializeInto->GetLength());

auto result = (AuxArray<int> \*)(deserializeInto->GetBuffer() + serialized->offset);

uint count = serialized->intCount;

result->count = count;

Assert(serialized->offset + result->GetDataSize() <= deserializeInto->GetLength());

auto content = (const byte\*)(serialized + 1);

auto current = content;

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

{

int32 value;

current = ReadConstantSizedInt32(current, &value);

result->elements[index] = value;

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int magicEnd;

current = ReadInt32(current, &magicEnd);

Assert(magicEnd == magicEndOfAuxIntArray);

#endif

return current;

}

const byte \*

DeserializeFloatArray(ScriptContext \* scriptContext, const byte \* buffer, ByteBlock \* deserializeInto)

{

auto serialized = (serialization\_alignment const SerializedFloatArray \*)buffer;

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

Assert(serialized->magic == magicStartOfAuxFltArray);

#endif

Assert(serialized->offset + sizeof(AuxArray<double>) < deserializeInto->GetLength());

auto result = (AuxArray<double> \*)(deserializeInto->GetBuffer() + serialized->offset);

uint count = serialized->floatCount;

result->count = count;

Assert(serialized->offset + result->GetDataSize() <= deserializeInto->GetLength());

auto content = (const byte\*)(serialized + 1);

auto current = content;

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

{

double value;

current = ReadDouble(current, &value);

result->elements[index] = value;

}

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int magicEnd;

current = ReadInt32(current, &magicEnd);

Assert(magicEnd == magicEndOfAuxFltArray);

#endif

return current;

}

};

// Construct the byte code cache. Copy things needed by inline 'Lookup' functions from reader.

ByteCodeCache::ByteCodeCache(ScriptContext \* scriptContext, ByteCodeBufferReader \* reader, int builtInPropertyCount)

: reader(reader), propertyCount(reader->string16Count), builtInPropertyCount(builtInPropertyCount)

{

auto alloc = scriptContext->SourceCodeAllocator();

propertyIds = AnewArray(alloc, PropertyId, propertyCount);

for (auto i=0; i < propertyCount; ++i)

{

propertyIds[i] = -1;

}

raw = reader->raw;

// Read and populate PropertyIds

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

{

PopulateLookupPropertyId(scriptContext, i);

}

}

// Deserialize and save a PropertyId

void ByteCodeCache::PopulateLookupPropertyId(ScriptContext \* scriptContext, int realOffset)

{

PropertyId idInCache = realOffset + this->builtInPropertyCount;

bool isPropertyRecord;

auto propertyName = reader->GetString16ById(idInCache, &isPropertyRecord);

if(isPropertyRecord)

{

auto propertyNameLength = reader->GetString16LengthById(idInCache);

const Js::PropertyRecord \* propertyRecord = scriptContext->GetThreadContext()->GetOrAddPropertyRecordBind(

JsUtil::CharacterBuffer<wchar\_t>(propertyName, propertyNameLength));

propertyIds[realOffset] = propertyRecord->GetPropertyId();

}

}

// Serialize function body

HRESULT ByteCodeSerializer::SerializeToBuffer(ScriptContext \* scriptContext, ArenaAllocator \* alloc, DWORD sourceByteLength, LPCUTF8 utf8Source, DWORD dwFunctionTableLength, BYTE \* functionTable, FunctionBody \* function, SRCINFO const\* srcInfo, bool allocateBuffer, byte \*\* buffer, DWORD \* bufferBytes, DWORD dwFlags)

{

int builtInPropertyCount = (dwFlags & GENERATE\_BYTE\_CODE\_BUFFER\_LIBRARY) != 0 ? PropertyIds::\_countJSOnlyProperty : TotalNumberOfBuiltInProperties;

Utf8SourceInfo \*utf8SourceInfo = function->GetUtf8SourceInfo();

HRESULT hr = utf8SourceInfo->EnsureLineOffsetCacheNoThrow();

if (FAILED(hr))

{

return hr;

}

int32 sourceCharLength = utf8SourceInfo->GetCchLength();

ByteCodeBufferBuilder builder(sourceByteLength, sourceCharLength, utf8Source, dwFunctionTableLength, functionTable, utf8SourceInfo, scriptContext, alloc, dwFlags, builtInPropertyCount);

hr = builder.AddTopFunctionBody(function, srcInfo);

if (SUCCEEDED(hr))

{

hr = builder.Create(allocateBuffer, buffer, bufferBytes);

}

#if INSTRUMENT\_BUFFER\_INTS

for (int i = 0; i < 4; i++)

{

printf("[BCGENSTATS] %d, %d\n", i, Counts[i]);

}

#endif

return hr;

}

HRESULT ByteCodeSerializer::DeserializeFromBuffer(ScriptContext \* scriptContext, ulong scriptFlags, LPCUTF8 utf8Source, SRCINFO const \* srcInfo, byte \* buffer, NativeModule \*nativeModule, FunctionBody\*\* function, uint sourceIndex)

{

return ByteCodeSerializer::DeserializeFromBufferInternal(scriptContext, scriptFlags, utf8Source, /\* sourceHolder \*/ nullptr, srcInfo, buffer, nativeModule, function, sourceIndex);

}

// Deserialize function body from supplied buffer

HRESULT ByteCodeSerializer::DeserializeFromBuffer(ScriptContext \* scriptContext, ulong scriptFlags, ISourceHolder\* sourceHolder, SRCINFO const \* srcInfo, byte \* buffer, NativeModule \*nativeModule, FunctionBody\*\* function, uint sourceIndex)

{

AssertMsg(sourceHolder != nullptr, "SourceHolder can't be null, if you have an empty source then pass ISourceHolder::GetEmptySourceHolder()");

return ByteCodeSerializer::DeserializeFromBufferInternal(scriptContext, scriptFlags, /\* utf8Source \*/ nullptr, sourceHolder, srcInfo, buffer, nativeModule, function, sourceIndex);

}

HRESULT ByteCodeSerializer::DeserializeFromBufferInternal(ScriptContext \* scriptContext, ulong scriptFlags, LPCUTF8 utf8Source, ISourceHolder\* sourceHolder, SRCINFO const \* srcInfo, byte \* buffer, NativeModule \*nativeModule, FunctionBody\*\* function, uint sourceIndex)

{

//ETW Event start

JS\_ETW(EventWriteJSCRIPT\_BYTECODEDESERIALIZE\_START(scriptContext, 0));

auto alloc = scriptContext->SourceCodeAllocator();

bool isLibraryCode = ((scriptFlags & fscrIsLibraryCode) == fscrIsLibraryCode);

int builtInPropertyCount = isLibraryCode ? PropertyIds::\_countJSOnlyProperty : TotalNumberOfBuiltInProperties;

auto reader = Anew(alloc, ByteCodeBufferReader, scriptContext, buffer, isLibraryCode, builtInPropertyCount);

auto hr = reader->ReadHeader();

if (FAILED(hr))

{

return hr;

}

ENTER\_PINNED\_SCOPE(Js::Utf8SourceInfo, sourceInfo);

ENTER\_PINNED\_SCOPE(SRCINFO const, pinnedSrcInfo);

pinnedSrcInfo = srcInfo;

if(sourceIndex == Js::Constants::InvalidSourceIndex)

{

if (sourceHolder == nullptr)

{

sourceHolder = utf8Source == nullptr ? ISourceHolder::GetEmptySourceHolder() : RecyclerNew(scriptContext->GetRecycler(), SimpleSourceHolder, utf8Source, reader->sourceSize);

}

sourceInfo = Js::Utf8SourceInfo::NewWithHolder(scriptContext, sourceHolder, reader->sourceCharLength, pinnedSrcInfo);

reader->utf8SourceInfo = sourceInfo;

reader->sourceIndex = scriptContext->SaveSourceNoCopy(sourceInfo, reader->sourceCharLength, false);

if(isLibraryCode)

{

sourceInfo->SetIsLibraryCode();

}

sourceInfo->CreateLineOffsetCache(reader->lineInfoCache, reader->lineInfoCacheCount);

}

else

{

Assert(CONFIG\_FLAG(ForceSerialized));

sourceInfo = scriptContext->GetSource(sourceIndex);

reader->utf8SourceInfo = sourceInfo;

reader->sourceIndex = sourceIndex;

}

auto cache = Anew(alloc, ByteCodeCache, scriptContext, reader, builtInPropertyCount);

hr = reader->ReadTopFunctionBody(function, sourceInfo, cache, ((scriptFlags & fscrAllowFunctionProxy) == fscrAllowFunctionProxy), nativeModule);

//ETW Event stop

JS\_ETW(EventWriteJSCRIPT\_BYTECODEDESERIALIZE\_STOP(scriptContext,0));

LEAVE\_PINNED\_SCOPE();

LEAVE\_PINNED\_SCOPE();

return hr;

}

void ByteCodeSerializer::ReadSourceInfo(const DeferDeserializeFunctionInfo\* deferredFunction, int& lineNumber, int& columnNumber, bool& m\_isEval, bool& m\_isDynamicFunction)

{

ByteCodeCache\* cache = deferredFunction->m\_cache;

ByteCodeBufferReader\* reader = cache->GetReader();

reader->ReadSourceInfo(deferredFunction->m\_functionBytes, lineNumber, columnNumber, m\_isEval, m\_isDynamicFunction);

}

FunctionBody\* ByteCodeSerializer::DeserializeFunction(ScriptContext\* scriptContext, DeferDeserializeFunctionInfo\* deferredFunction)

{

FunctionBody\* deserializedFunctionBody = nullptr;

ByteCodeCache\* cache = deferredFunction->m\_cache;

ByteCodeBufferReader\* reader = cache->GetReader();

HRESULT hr = reader->ReadFunctionBody(deferredFunction->m\_functionBytes, (FunctionProxy \*\*)&deserializedFunctionBody, deferredFunction->m\_utf8SourceInfo, cache, deferredFunction->m\_nativeModule, true /\* deserialize this \*/, false /\* deserialize nested functions \*/, deferredFunction);

if (FAILED(hr))

{

// This should never happen as the code is currently

// structured since we validate the serialized bytecode during creation

// of function proxies. In the future though, when we reorganize the byte

// code file format, we could hit this error, in which case we

// need a strategy to deal with this.

Assert(false);

Js::Throw::InternalError();

}

return deserializedFunctionBody;

}

SerializedAuxiliary::SerializedAuxiliary( uint offset, SerializedAuxiliaryKind kind ) :

offset(offset), kind(kind)

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

, auxMagic(magicStartOfAux)

#endif

{

}

SerializedVarArray::SerializedVarArray( uint offset, bool isVarCount, int varCount ) :

SerializedAuxiliary(offset, isVarCount ? sakVarArrayVarCount : sakVarArrayIntCount), varCount(varCount)

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

, magic(magicStartOfAuxVarArray)

#endif

{

}

SerializedIntArray::SerializedIntArray( uint offset, int intCount ) :

SerializedAuxiliary(offset, sakIntArray), intCount(intCount)

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

, magic(magicStartOfAuxIntArray)

#endif

{

}

SerializedFloatArray::SerializedFloatArray( uint offset, int floatCount ) :

SerializedAuxiliary(offset, sakFloatArray), floatCount(floatCount)

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

, magic(magicStartOfAuxFltArray)

#endif

{

}

SerializedPropertyIdArray::SerializedPropertyIdArray( uint offset, int propertyCount, int extraSlots, bool hadDuplicates, bool has\_\_proto\_\_) :

SerializedAuxiliary(offset, sakPropertyIdArray), propertyCount(propertyCount), extraSlots(extraSlots), hadDuplicates(hadDuplicates), has\_\_proto\_\_(has\_\_proto\_\_)

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

, magic(magicStartOfAuxPropIdArray)

#endif

{

}

SerializedFuncInfoArray::SerializedFuncInfoArray( uint offset, int count ) :

SerializedAuxiliary(offset, sakFuncInfoArray), count(count)

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

, magic(magicStartOfAuxFuncInfoArray)

#endif

{

}

}

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

// 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

// Turn this on to enable magic constants in byte code (useful for debugging)

//#define BYTE\_CODE\_MAGIC\_CONSTANTS

#include "ByteCode\ByteCodeSerializeFlags.h"

namespace Js

{

// Some things are obscured by xor. This helps catch cases in which, for example, indirect property ids

// are mistakenly mixed with actual property IDs.

#if DBG & VALIDATE\_SERIALIZED\_BYTECODE

#define SERIALIZER\_OBSCURE\_PROPERTY\_ID 0xdef00000

#define SERIALIZER\_OBSCURE\_NONBUILTIN\_PROPERTY\_ID 0xdeb00000

#define SERIALIZER\_OBSCURE\_LITERAL\_OBJECT\_ID 0xded00000

#else

#define SERIALIZER\_OBSCURE\_PROPERTY\_ID 0x00000000

#define SERIALIZER\_OBSCURE\_NONBUILTIN\_PROPERTY\_ID 0x00000000

#define SERIALIZER\_OBSCURE\_LITERAL\_OBJECT\_ID 0x00000000

#endif

class ByteCodeBufferReader;

enum SerializedAuxiliaryKind : byte

{

sakVarArrayIntCount = 1,

sakVarArrayVarCount = 2,

sakPropertyIdArray = 3,

sakFuncInfoArray = 4,

sakIntArray = 5,

sakFloatArray = 6,

sakPropertyIdArrayForCachedScope = 7, // serialization only type, indiciate extra slots

};

// Tightly pack serialized structures

#pragma pack(push, 1)

// Describes the kind of auxiliary

struct SerializedAuxiliary

{

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int auxMagic; // magicStartOfAux

#endif

uint offset;

SerializedAuxiliaryKind kind;

SerializedAuxiliary(uint offset, SerializedAuxiliaryKind kind);

};

// The in-memory layout of the serialized analog of VarArray

struct SerializedVarArray : SerializedAuxiliary

{

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int magic; // magicStartOfAuxVarArray

#endif

int varCount;

SerializedVarArray(uint offset, bool isVarCount, int varCount);

};

struct SerializedIntArray : SerializedAuxiliary

{

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int magic; // magicStartOfAuxIntArray

#endif

int intCount;

SerializedIntArray(uint offset, int intCount);

};

struct SerializedFloatArray : SerializedAuxiliary

{

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int magic; // magicStartOfAuxFltArray

#endif

int floatCount;

SerializedFloatArray(uint offset, int floatCount);

};

// The in-memory layout of the serialized analog of PropertyIdArray

struct SerializedPropertyIdArray : SerializedAuxiliary

{

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int magic; // magicStartOfAuxPropIdArray

#endif

int propertyCount;

int extraSlots;

bool hadDuplicates;

bool has\_\_proto\_\_;

SerializedPropertyIdArray(uint offset, int propertyCount, int extraSlots, bool hadDuplicates, bool has\_\_proto\_\_);

};

// The in-memory layout of the serialized analog of FuncInfoArray

struct SerializedFuncInfoArray : SerializedAuxiliary

{

#ifdef BYTE\_CODE\_MAGIC\_CONSTANTS

int magic; // magicStartOfAuxFuncInfoArray

#endif

int count;

SerializedFuncInfoArray(uint offset, int count);

};

#pragma pack(pop)

// Holds information about the deserialized bytecode cache. Contains fast inline functions

// for the lookup hit case. The slower deserialization of VarArray, etc are in the .cpp.

class ByteCodeCache

{

ByteCodeBufferReader \* reader;

const byte \* raw;

PropertyId \* propertyIds;

int propertyCount;

int builtInPropertyCount;

public:

ByteCodeCache(ScriptContext \* scriptContext, ByteCodeBufferReader \* reader, int builtInPropertyCount);

void PopulateLookupPropertyId(ScriptContext \* scriptContext, int realArrayOffset);

ByteCodeBufferReader\* GetReader()

{

return reader;

}

// Convert a serialized propertyID into a real one.

inline PropertyId LookupPropertyId(PropertyId obscuredIdInCache) const

{

auto unobscured = obscuredIdInCache ^ SERIALIZER\_OBSCURE\_PROPERTY\_ID;

if (unobscured < builtInPropertyCount || unobscured==/\*nil\*/0xffffffff)

{

return unobscured; // This is a built in property id

}

auto realOffset = unobscured - builtInPropertyCount;

Assert(realOffset<propertyCount);

Assert(propertyIds[realOffset]!=-1);

return propertyIds[realOffset];

}

// Convert a serialized propertyID into a real one.

inline PropertyId LookupNonBuiltinPropertyId(PropertyId obscuredIdInCache) const

{

auto realOffset = obscuredIdInCache ^ SERIALIZER\_OBSCURE\_NONBUILTIN\_PROPERTY\_ID;

Assert(realOffset<propertyCount);

Assert(propertyIds[realOffset]!=-1);

return propertyIds[realOffset];

}

// Get the raw byte code buffer.

inline const byte \* GetBuffer() const

{

return raw;

}

};

// Methods for serializing and deserializing function bodies.

struct ByteCodeSerializer

{

// Serialize a function body.

static HRESULT SerializeToBuffer(ScriptContext \* scriptContext, ArenaAllocator \* alloc, DWORD sourceCodeLength, LPCUTF8 utf8Source, DWORD dwFunctionTableLength, BYTE \* functionTable, FunctionBody \* function, SRCINFO const\* srcInfo, bool allocateBuffer, byte \*\* buffer, DWORD \* bufferBytes, DWORD dwFlags = 0);

// Deserialize a function body. The content of utf8Source must be the same as was originally passed to SerializeToBuffer

static HRESULT DeserializeFromBuffer(ScriptContext \* scriptContext, ulong scriptFlags, LPCUTF8 utf8Source, SRCINFO const \* srcInfo, byte \* buffer, NativeModule \*nativeModule, FunctionBody\*\* function, uint sourceIndex = Js::Constants::InvalidSourceIndex);

static HRESULT DeserializeFromBuffer(ScriptContext \* scriptContext, ulong scriptFlags, ISourceHolder\* sourceHolder, SRCINFO const \* srcInfo, byte \* buffer, NativeModule \*nativeModule, FunctionBody\*\* function, uint sourceIndex = Js::Constants::InvalidSourceIndex);

static FunctionBody\* DeserializeFunction(ScriptContext\* scriptContext, DeferDeserializeFunctionInfo\* deferredFunction);

// This lib doesn't directly depend on the generated interfaces. Ensure the same codes with a C\_ASSERT

static const HRESULT CantGenerate = 0x80020201L;

static const HRESULT InvalidByteCode = 0x80020202L;

static void ReadSourceInfo(const DeferDeserializeFunctionInfo\* deferredFunction, int& lineNumber, int& columnNumber, bool& m\_isEval, bool& m\_isDynamicFunction);

private:

static HRESULT DeserializeFromBufferInternal(ScriptContext \* scriptContext, ulong scriptFlags, LPCUTF8 utf8Source, ISourceHolder\* sourceHolder, SRCINFO const \* srcInfo, byte \* buffer, NativeModule \*nativeModule, FunctionBody\*\* function, uint sourceIndex = Js::Constants::InvalidSourceIndex);

};

}

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

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

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

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

#include "RuntimeByteCodePch.h"

namespace Js

{

void ByteCodeWriter::Create()

{

m\_loopNest = 0;

m\_byteCodeCount = 0;

m\_byteCodeWithoutLDACount = 0;

m\_byteCodeInLoopCount = 0;

m\_functionWrite = nullptr;

m\_pMatchingNode = nullptr;

m\_matchingNodeRefCount = 0;

m\_tmpRegCount = 0;

DebugOnly(isInitialized = false);

DebugOnly(isInUse = false);

}

void ByteCodeWriter::InitData(ArenaAllocator\* alloc, long initCodeBufferSize)

{

Assert(!isInUse);

Assert(!isInitialized);

DebugOnly(isInitialized = true);

m\_labelOffsets = JsUtil::List<uint, ArenaAllocator>::New(alloc);

m\_jumpOffsets = JsUtil::List<JumpInfo, ArenaAllocator>::New(alloc);

m\_loopHeaders = JsUtil::List<LoopHeaderData, ArenaAllocator>::New(alloc);

m\_byteCodeData.Create(initCodeBufferSize, alloc);

m\_subexpressionNodesStack = Anew(alloc, JsUtil::Stack<SubexpressionNode>, alloc);

// These data units have exponential growth strategy - let's start small and grow them

m\_auxiliaryData.Create(256, alloc);

m\_auxContextData.Create(256, alloc);

callRegToLdFldCacheIndexMap = Anew(alloc, CallRegToLdFldCacheIndexMap,

alloc,

17);

#ifdef BYTECODE\_BRANCH\_ISLAND

useBranchIsland = true;

inEnsureLongBranch = false;

lastOpcode = Js::OpCode::FunctionEntry;

this->UpdateNextBranchIslandOffset(0, 0);

m\_longJumpOffsets = JsUtil::List<JumpInfo, ArenaAllocator>::New(alloc);

#endif

}

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

///

/// Begin() configures this instance to generate byte-code for a specific

/// JavascriptFunction:

///

/// - Byte-code will be written until the caller uses End() to close and commit

/// the stream to the given function, or Reset() to discard and reset to an

/// empty state.

///

/// - Each ByteCodeWriter may be used multiple times, but may only generate a

/// single byte-code stream for a single function at a time.

///

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

void ByteCodeWriter::Begin(ByteCodeGenerator\* byteCodeGenerator, FunctionBody\* functionWrite, ArenaAllocator\* alloc, bool doJitLoopBodies, bool hasLoop)

{

Assert(!isInUse);

AssertMsg(m\_functionWrite == nullptr, "Cannot nest Begin() calls");

AssertMsg(functionWrite != nullptr, "Must have valid function to write");

AssertMsg(functionWrite->GetByteCode() == nullptr, "Function should not already have a byte-code body");

AssertMsg(functionWrite->GetLocalsCount() > 0, "Must always have R0 for return-value");

DebugOnly(isInUse = true);

m\_functionWrite = functionWrite;

m\_doJitLoopBodies = doJitLoopBodies;

m\_doInterruptProbe = functionWrite->GetScriptContext()->GetThreadContext()->DoInterruptProbe(functionWrite);

m\_hasLoop = hasLoop;

m\_isInDebugMode = byteCodeGenerator->IsInDebugMode();

}

template <typename T>

void ByteCodeWriter::PatchJumpOffset(JsUtil::List<JumpInfo, ArenaAllocator> \* jumpOffset, byte \* byteBuffer, uint byteCount)

{

jumpOffset->Map([=](int index, JumpInfo& jumpInfo)

{

//

// Read "labelID" stored at the offset within the byte-code.

//

uint jumpByteOffset = jumpInfo.patchOffset;

AssertMsg(jumpByteOffset < byteCount - sizeof(T),

"Must have valid jump site within byte-code to back-patch");

unaligned T \* pnBackPatch = reinterpret\_cast<unaligned T \*>(&byteBuffer[jumpByteOffset]);

ByteCodeLabel labelID = jumpInfo.labelId;

CheckLabel(labelID);

uint offsetToEndOfLayoutByteSize = \*pnBackPatch;

Assert(offsetToEndOfLayoutByteSize < 0x20);

//

// Use "labelID" to lookup the destination offset, replacing the temporary data in the

// byte-code.

//

uint labelByteOffset = m\_labelOffsets->Item(labelID);

AssertMsg(labelByteOffset != UINT\_MAX, "ERROR: Destination labels must be marked before closing");

int relativeJumpOffset = labelByteOffset - jumpByteOffset - offsetToEndOfLayoutByteSize;

#ifdef BYTECODE\_BRANCH\_ISLAND

Assert(!useBranchIsland || (jumpOffset != m\_jumpOffsets || (relativeJumpOffset < GetBranchLimit() && relativeJumpOffset >= -GetBranchLimit())));

#endif

Assert((T)relativeJumpOffset == relativeJumpOffset);

\*pnBackPatch = (T)relativeJumpOffset;

});

}

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

///

/// End() completes generating byte-code for the given JavascriptFunction and

/// commits it to the function's body.

///

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

#ifdef LOG\_BYTECODE\_AST\_RATIO

void ByteCodeWriter::End(long currentAstSize, long maxAstSize)

#else

void ByteCodeWriter::End()

#endif

{

Assert(isInUse);

CheckOpen();

Empty(OpCode::EndOfBlock);

ByteBlock\* finalByteCodeBlock;

ScriptContext\* scriptContext = m\_functionWrite->GetScriptContext();

m\_byteCodeData.Copy(scriptContext->GetRecycler(), &finalByteCodeBlock);

byte \* byteBuffer = finalByteCodeBlock->GetBuffer();

uint byteCount = m\_byteCodeData.GetCurrentOffset();

//

// Update all branch targets with their actual label destinations.

//

#ifdef BYTECODE\_BRANCH\_ISLAND

if (useBranchIsland)

{

PatchJumpOffset<JumpOffset>(m\_jumpOffsets, byteBuffer, byteCount);

PatchJumpOffset<LongJumpOffset>(m\_longJumpOffsets, byteBuffer, byteCount);

}

else

{

PatchJumpOffset<LongJumpOffset>(m\_jumpOffsets, byteBuffer, byteCount);

}

#else

PatchJumpOffset<JumpOffset>(m\_jumpOffsets, byteBuffer, byteCount);

#endif

// Patch up the root object load inline cache with the start index

uint rootObjectLoadInlineCacheStart = this->m\_functionWrite->GetRootObjectLoadInlineCacheStart();

rootObjectLoadInlineCacheOffsets.Map([=](size\_t offset)

{

Assert(offset < byteCount - sizeof(int));

unaligned uint \* pnBackPatch = reinterpret\_cast<unaligned uint \*>(&byteBuffer[offset]);

\*pnBackPatch += rootObjectLoadInlineCacheStart;

});

// Patch up the root object load method inline cache with the start index

uint rootObjectLoadMethodInlineCacheStart = this->m\_functionWrite->GetRootObjectLoadMethodInlineCacheStart();

rootObjectLoadMethodInlineCacheOffsets.Map([=](size\_t offset)

{

Assert(offset < byteCount - sizeof(int));

unaligned uint \* pnBackPatch = reinterpret\_cast<unaligned uint \*>(&byteBuffer[offset]);

\*pnBackPatch += rootObjectLoadMethodInlineCacheStart;

});

// Patch up the root object store inline cache with the start index

uint rootObjectStoreInlineCacheStart = this->m\_functionWrite->GetRootObjectStoreInlineCacheStart();

rootObjectStoreInlineCacheOffsets.Map([=](size\_t offset)

{

Assert(offset < byteCount - sizeof(int));

unaligned uint \* pnBackPatch = reinterpret\_cast<unaligned uint \*>(&byteBuffer[offset]);

\*pnBackPatch += rootObjectStoreInlineCacheStart;

});

//

// Store the final trimmed byte-code on the function.

//

ByteBlock\* finalAuxiliaryBlock;

ByteBlock\* finalAuxiliaryContextBlock;

m\_auxiliaryData.Copy(m\_functionWrite->GetScriptContext()->GetRecycler(), &finalAuxiliaryBlock);

m\_auxContextData.Copy(m\_functionWrite->GetScriptContext()->GetRecycler(), &finalAuxiliaryContextBlock);

m\_functionWrite->AllocateInlineCache();

m\_functionWrite->AllocateObjectLiteralTypeArray();

if (!PHASE\_OFF(Js::ScriptFunctionWithInlineCachePhase, m\_functionWrite) && !PHASE\_OFF(Js::InlineApplyTargetPhase, m\_functionWrite))

{

if (m\_functionWrite->CanFunctionObjectHaveInlineCaches())

{

m\_functionWrite->SetInlineCachesOnFunctionObject(true);

}

}

if (this->DoJitLoopBodies() &&

!this->m\_functionWrite->GetFunctionBody()->GetHasFinally() &&

!(this->m\_functionWrite->GetFunctionBody()->GetHasTry() && PHASE\_OFF(Js::JITLoopBodyInTryCatchPhase, this->m\_functionWrite)))

{

AllocateLoopHeaders();

}

m\_functionWrite->MarkScript(finalByteCodeBlock, finalAuxiliaryBlock, finalAuxiliaryContextBlock,

m\_byteCodeCount, m\_byteCodeInLoopCount, m\_byteCodeWithoutLDACount);

#if ENABLE\_PROFILE\_INFO

m\_functionWrite->LoadDynamicProfileInfo();

#endif

JS\_ETW(EventWriteJSCRIPT\_BYTECODEGEN\_METHOD(m\_functionWrite->GetHostSourceContext(), m\_functionWrite->GetScriptContext(), m\_functionWrite->GetLocalFunctionId(), m\_functionWrite->GetByteCodeCount(), this->GetTotalSize(), m\_functionWrite->GetExternalDisplayName()));

#ifdef LOG\_BYTECODE\_AST\_RATIO

// log the bytecode AST ratio

if (currentAstSize == maxAstSize)

{

float astBytecodeRatio = (float)currentAstSize / (float)byteCount;

Output::Print(L"\tAST Bytecode ratio: %f\n", astBytecodeRatio);

}

#endif

// TODO: add validation for source mapping under #dbg

//

// Reset the writer to prepare for the next user.

//

Reset();

}

void ByteCodeWriter::AllocateLoopHeaders()

{

m\_functionWrite->AllocateLoopHeaders();

m\_loopHeaders->Map([this](int index, ByteCodeWriter::LoopHeaderData& data)

{

LoopHeader \*loopHeader = m\_functionWrite->GetLoopHeader(index);

loopHeader->startOffset = data.startOffset;

loopHeader->endOffset = data.endOffset;

loopHeader->isNested = data.isNested;

});

}

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

///

/// Reset() discards any current byte-code and resets to a known "empty" state:

/// - This method may be called at any time between Create() and Dispose().

///

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

void ByteCodeWriter::Reset()

{

DebugOnly(isInUse = false);

Assert(isInitialized);

m\_byteCodeData.Reset();

m\_auxiliaryData.Reset();

m\_auxContextData.Reset();

#ifdef BYTECODE\_BRANCH\_ISLAND

lastOpcode = Js::OpCode::FunctionEntry;

this->UpdateNextBranchIslandOffset(0, 0);

m\_longJumpOffsets->Clear();

#endif

m\_labelOffsets->Clear();

m\_jumpOffsets->Clear();

m\_loopHeaders->Clear();

rootObjectLoadInlineCacheOffsets.Clear(m\_labelOffsets->GetAllocator());

rootObjectStoreInlineCacheOffsets.Clear(m\_labelOffsets->GetAllocator());

rootObjectLoadMethodInlineCacheOffsets.Clear(m\_labelOffsets->GetAllocator());

callRegToLdFldCacheIndexMap->ResetNoDelete();

m\_pMatchingNode = nullptr;

m\_matchingNodeRefCount = 0;

m\_functionWrite = nullptr;

m\_byteCodeCount = 0;

m\_byteCodeWithoutLDACount = 0;

m\_byteCodeInLoopCount = 0;

m\_loopNest = 0;

m\_currentDebuggerScope = nullptr;

}

inline Js::RegSlot ByteCodeWriter::ConsumeReg(Js::RegSlot reg)

{

CheckReg(reg);

Assert(this->m\_functionWrite);

return this->m\_functionWrite->MapRegSlot(reg);

}

inline void ByteCodeWriter::CheckOpen()

{

AssertMsg(m\_functionWrite != nullptr, "Must Begin() a function to write byte-code into");

}

inline void ByteCodeWriter::CheckOp(OpCode op, OpLayoutType layoutType)

{

AssertMsg(OpCodeUtil::IsValidByteCodeOpcode(op), "Ensure valid OpCode");

#if ENABLE\_NATIVE\_CODEGEN

AssertMsg(!OpCodeAttr::BackEndOnly(op), "Can't write back end only OpCode");

#endif

AssertMsg(OpCodeUtil::GetOpCodeLayout(op) == layoutType, "Ensure correct layout for OpCode");

}

inline void ByteCodeWriter::CheckLabel(ByteCodeLabel labelID)

{

AssertMsg(labelID < m\_labelOffsets->Count(),

"Label must be previously defined before being marked in the byte-code");

}

inline void ByteCodeWriter::CheckReg(RegSlot registerID)

{

AssertMsg(registerID != Js::Constants::NoRegister, "bad register");

if (registerID == Js::Constants::NoRegister)

Js::Throw::InternalError();

}

void ByteCodeWriter::Empty(OpCode op)

{

CheckOpen();

CheckOp(op, OpLayoutType::Empty);

m\_byteCodeData.Encode(op, this);

}

#define MULTISIZE\_LAYOUT\_WRITE(layout, ...) \

if (!TryWrite##layout<SmallLayoutSizePolicy>(\_\_VA\_ARGS\_\_) && !TryWrite##layout<MediumLayoutSizePolicy>(\_\_VA\_ARGS\_\_)) \

{ \

bool success = TryWrite##layout<LargeLayoutSizePolicy>(\_\_VA\_ARGS\_\_); \

Assert(success); \

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteReg1(OpCode op, RegSlot R0)

{

OpLayoutT\_Reg1<SizePolicy> layout;

if (SizePolicy::Assign(layout.R0, R0))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::Reg1(OpCode op, RegSlot R0)

{

CheckOpen();

CheckOp(op, OpLayoutType::Reg1);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

R0 = ConsumeReg(R0);

MULTISIZE\_LAYOUT\_WRITE(Reg1, op, R0);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteReg2WithICIndex(OpCode op, RegSlot R0, RegSlot R1, uint32 inlineCacheIndex, bool isRootLoad)

{

OpLayoutT\_Reg2WithICIndex<SizePolicy> layout;

if (SizePolicy::Assign(layout.R0, R0) && SizePolicy::Assign(layout.R1, R1) && SizePolicy::Assign(layout.inlineCacheIndex, inlineCacheIndex))

{

size\_t offset = m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

if (isRootLoad)

{

size\_t inlineCacheOffset = offset + OpCodeUtil::EncodedSize(op, SizePolicy::LayoutEnum)

+ offsetof(OpLayoutT\_Reg2WithICIndex<SizePolicy>, inlineCacheIndex);

rootObjectLoadMethodInlineCacheOffsets.Prepend(m\_labelOffsets->GetAllocator(), inlineCacheOffset);

}

return true;

}

return false;

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteReg2(OpCode op, RegSlot R0, RegSlot R1)

{

OpLayoutT\_Reg2<SizePolicy> layout;

if (SizePolicy::Assign(layout.R0, R0) && SizePolicy::Assign(layout.R1, R1))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::Reg2(OpCode op, RegSlot R0, RegSlot R1)

{

CheckOpen();

CheckOp(op, OpLayoutType::Reg2);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

if (DoDynamicProfileOpcode(CheckThisPhase) ||

DoDynamicProfileOpcode(TypedArrayTypeSpecPhase) ||

DoDynamicProfileOpcode(ArrayCheckHoistPhase))

{

if (op == OpCode::StrictLdThis)

{

op = OpCode::ProfiledStrictLdThis;

}

}

R0 = ConsumeReg(R0);

R1 = ConsumeReg(R1);

CacheIdUnit unit;

unit.cacheId = Js::Constants::NoInlineCacheIndex;

callRegToLdFldCacheIndexMap->TryGetValueAndRemove(R1, &unit);

bool isProfiled = false;

bool isProfiled2 = false;

bool isReg2WithICIndex = false;

Js::ProfileId profileId = Js::Constants::NoProfileId;

Js::ProfileId profileId2 = Js::Constants::NoProfileId;

if (op == Js::OpCode::BeginSwitch && DoDynamicProfileOpcode(SwitchOptPhase) &&

this->m\_functionWrite->AllocProfiledSwitch(&profileId))

{

OpCodeUtil::ConvertNonCallOpToProfiled(op);

isProfiled = true;

}

Assert(DoProfileNewScObjArrayOp(op) == false);

Assert(DoProfileNewScObjectOp(op) == false);

if (op == Js::OpCode::LdLen\_A

&& (DoDynamicProfileOpcode(AggressiveIntTypeSpecPhase) ||

DoDynamicProfileOpcode(FloatTypeSpecPhase) ||

DoDynamicProfileOpcode(TypedArrayTypeSpecPhase) ||

DoDynamicProfileOpcode(ArrayCheckHoistPhase))

&& this->m\_functionWrite->AllocProfiledLdElemId(&profileId))

{

OpCodeUtil::ConvertNonCallOpToProfiled(op);

isProfiled = true;

}

if (isReg2WithICIndex)

{

MULTISIZE\_LAYOUT\_WRITE(Reg2WithICIndex, op, R0, R1, unit.cacheId, unit.isRootObjectCache);

}

else

{

MULTISIZE\_LAYOUT\_WRITE(Reg2, op, R0, R1);

}

if (isProfiled)

{

m\_byteCodeData.Encode(&profileId, sizeof(Js::ProfileId));

if (isProfiled2)

{

m\_byteCodeData.Encode(&profileId2, sizeof(Js::ProfileId));

}

}

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteReg3(OpCode op, RegSlot R0, RegSlot R1, RegSlot R2)

{

OpLayoutT\_Reg3<SizePolicy> layout;

if (SizePolicy::Assign(layout.R0, R0) && SizePolicy::Assign(layout.R1, R1) && SizePolicy::Assign(layout.R2, R2))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::Reg3(OpCode op, RegSlot R0, RegSlot R1, RegSlot R2)

{

CheckOpen();

CheckOp(op, OpLayoutType::Reg3);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

R0 = ConsumeReg(R0);

R1 = ConsumeReg(R1);

R2 = ConsumeReg(R2);

ProfileId profileId = 0;

bool isProfiled = false;

if ((DoDynamicProfileOpcode(FloatTypeSpecPhase) && (op == Js::OpCode::Div\_A || op == Js::OpCode::Rem\_A)) &&

this->m\_functionWrite->AllocProfiledDivOrRem(&profileId))

{

isProfiled = true;

OpCodeUtil::ConvertNonCallOpToProfiled(op);

}

MULTISIZE\_LAYOUT\_WRITE(Reg3, op, R0, R1, R2);

if (isProfiled)

{

m\_byteCodeData.Encode(&profileId, sizeof(Js::ProfileId));

}

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteReg3C(OpCode op, RegSlot R0, RegSlot R1, RegSlot R2, CacheId cacheId)

{

OpLayoutT\_Reg3C<SizePolicy> layout;

if (SizePolicy::Assign(layout.R0, R0) && SizePolicy::Assign(layout.R1, R1) && SizePolicy::Assign(layout.R2, R2)

&& SizePolicy::Assign(layout.inlineCacheIndex, cacheId))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::Reg3C(OpCode op, RegSlot R0, RegSlot R1, RegSlot R2, uint cacheId)

{

CheckOpen();

CheckOp(op, OpLayoutType::Reg3C);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

R0 = ConsumeReg(R0);

R1 = ConsumeReg(R1);

R2 = ConsumeReg(R2);

MULTISIZE\_LAYOUT\_WRITE(Reg3C, op, R0, R1, R2, cacheId);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteReg4(OpCode op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3)

{

OpLayoutT\_Reg4<SizePolicy> layout;

if (SizePolicy::Assign(layout.R0, R0) && SizePolicy::Assign(layout.R1, R1) && SizePolicy::Assign(layout.R2, R2)

&& SizePolicy::Assign(layout.R3, R3))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::Reg4(OpCode op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3)

{

CheckOpen();

CheckOp(op, OpLayoutType::Reg4);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

R0 = ConsumeReg(R0);

R1 = ConsumeReg(R1);

R2 = ConsumeReg(R2);

R3 = ConsumeReg(R3);

MULTISIZE\_LAYOUT\_WRITE(Reg4, op, R0, R1, R2, R3);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteReg2B1(OpCode op, RegSlot R0, RegSlot R1, uint8 B2)

{

OpLayoutT\_Reg2B1<SizePolicy> layout;

if (SizePolicy::Assign(layout.R0, R0) && SizePolicy::Assign(layout.R1, R1) && SizePolicy::Assign(layout.B2, B2))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::Reg2B1(OpCode op, RegSlot R0, RegSlot R1, uint8 B2)

{

CheckOpen();

CheckOp(op, OpLayoutType::Reg2B1);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

R0 = ConsumeReg(R0);

R1 = ConsumeReg(R1);

MULTISIZE\_LAYOUT\_WRITE(Reg2B1, op, R0, R1, B2);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteReg3B1(OpCode op, RegSlot R0, RegSlot R1, RegSlot R2, uint8 B3)

{

OpLayoutT\_Reg3B1<SizePolicy> layout;

if (SizePolicy::Assign(layout.R0, R0) && SizePolicy::Assign(layout.R1, R1) && SizePolicy::Assign(layout.R2, R2)

&& SizePolicy::Assign(layout.B3, B3))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::Reg3B1(OpCode op, RegSlot R0, RegSlot R1, RegSlot R2, uint8 B3)

{

CheckOpen();

CheckOp(op, OpLayoutType::Reg3B1);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

R0 = ConsumeReg(R0);

R1 = ConsumeReg(R1);

R2 = ConsumeReg(R2);

MULTISIZE\_LAYOUT\_WRITE(Reg3B1, op, R0, R1, R2, B3);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteReg5(OpCode op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3, RegSlot R4)

{

OpLayoutT\_Reg5<SizePolicy> layout;

if (SizePolicy::Assign(layout.R0, R0) && SizePolicy::Assign(layout.R1, R1) && SizePolicy::Assign(layout.R2, R2)

&& SizePolicy::Assign(layout.R3, R3) && SizePolicy::Assign(layout.R4, R4))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::Reg5(OpCode op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3, RegSlot R4)

{

CheckOpen();

CheckOp(op, OpLayoutType::Reg5);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

R0 = ConsumeReg(R0);

R1 = ConsumeReg(R1);

R2 = ConsumeReg(R2);

R3 = ConsumeReg(R3);

R4 = ConsumeReg(R4);

MULTISIZE\_LAYOUT\_WRITE(Reg5, op, R0, R1, R2, R3, R4);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteUnsigned1(OpCode op, uint C1)

{

OpLayoutT\_Unsigned1<SizePolicy> layout;

if (SizePolicy::Assign(layout.C1, C1))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::Unsigned1(OpCode op, uint C1)

{

CheckOpen();

CheckOp(op, OpLayoutType::Unsigned1);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

MULTISIZE\_LAYOUT\_WRITE(Unsigned1, op, C1);

}

void ByteCodeWriter::ArgIn0(RegSlot reg)

{

AssertMsg(0 < m\_functionWrite->GetInParamsCount(),

"Ensure source arg was declared in prologue");

Reg1(OpCode::ArgIn0, reg);

}

template void ByteCodeWriter::ArgOut<true>(ArgSlot arg, RegSlot reg, ProfileId callSiteId);

template void ByteCodeWriter::ArgOut<false>(ArgSlot arg, RegSlot reg, ProfileId callSiteId);

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteArg(OpCode op, ArgSlot arg, RegSlot reg)

{

OpLayoutT\_Arg<SizePolicy> layout;

if (SizePolicy::Assign(layout.Arg, arg) && SizePolicy::Assign(layout.Reg, reg))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

template <bool isVar>

void ByteCodeWriter::ArgOut(ArgSlot arg, RegSlot reg, ProfileId callSiteId)

{

CheckOpen();

Assert(OpCodeAttr::HasMultiSizeLayout(OpCode::ArgOut\_A) && OpCodeAttr::HasMultiSizeLayout(OpCode::ArgOut\_ANonVar));

// Note: don't "consume" the arg slot, as the passed-in value is the final one.

reg = ConsumeReg(reg);

OpCode op;

if (isVar)

{

op = OpCode::ArgOut\_A;

}

else

{

op = OpCode::ArgOut\_ANonVar;

MULTISIZE\_LAYOUT\_WRITE(Arg, op, arg, reg);

return;

}

if (DoDynamicProfileOpcode(InlinePhase)

&& arg > 0 && arg < Js::Constants::MaximumArgumentCountForConstantArgumentInlining

&& (reg > FunctionBody::FirstRegSlot && reg < m\_functionWrite->GetConstantCount())

&& callSiteId != Js::Constants::NoProfileId

&& !m\_isInDebugMode // We don't inline in debug mode, so no need to emit ProfiledArgOut\_A

)

{

Assert((reg > FunctionBody::FirstRegSlot && reg < m\_functionWrite->GetConstantCount()));

MULTISIZE\_LAYOUT\_WRITE(Arg, Js::OpCode::ProfiledArgOut\_A, arg, reg);

m\_byteCodeData.Encode(&callSiteId, sizeof(Js::ProfileId));

}

else

{

MULTISIZE\_LAYOUT\_WRITE(Arg, op, arg, reg);

return;

}

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteArgNoSrc(OpCode op, ArgSlot arg)

{

OpLayoutT\_ArgNoSrc<SizePolicy> layout;

if (SizePolicy::Assign(layout.Arg, arg))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::ArgOutEnv(ArgSlot arg)

{

CheckOpen();

Assert(OpCodeAttr::HasMultiSizeLayout(OpCode::ArgOut\_Env));

MULTISIZE\_LAYOUT\_WRITE(ArgNoSrc, OpCode::ArgOut\_Env, arg);

}

void ByteCodeWriter::Br(ByteCodeLabel labelID)

{

Br(OpCode::Br, labelID);

}

// For switch case - default branching

void ByteCodeWriter::Br(OpCode op, ByteCodeLabel labelID)

{

CheckOpen();

CheckOp(op, OpLayoutType::Br);

CheckLabel(labelID);

Assert(!OpCodeAttr::HasMultiSizeLayout(op));

size\_t const offsetOfRelativeJumpOffsetFromEnd = sizeof(OpLayoutBr) - offsetof(OpLayoutBr, RelativeJumpOffset);

OpLayoutBr data;

data.RelativeJumpOffset = offsetOfRelativeJumpOffsetFromEnd;

m\_byteCodeData.Encode(op, &data, sizeof(data), this);

AddJumpOffset(op, labelID, offsetOfRelativeJumpOffsetFromEnd);

}

void ByteCodeWriter::BrS(OpCode op, ByteCodeLabel labelID, byte val)

{

CheckOpen();

CheckOp(op, OpLayoutType::BrS);

CheckLabel(labelID);

Assert(!OpCodeAttr::HasMultiSizeLayout(op));

size\_t const offsetOfRelativeJumpOffsetFromEnd = sizeof(OpLayoutBrS) - offsetof(OpLayoutBrS, RelativeJumpOffset);

OpLayoutBrS data;

data.RelativeJumpOffset = offsetOfRelativeJumpOffsetFromEnd;

data.val = val;

m\_byteCodeData.Encode(op, &data, sizeof(data), this);

AddJumpOffset(op, labelID, offsetOfRelativeJumpOffsetFromEnd);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteBrReg1(OpCode op, ByteCodeLabel labelID, RegSlot R1)

{

OpLayoutT\_BrReg1<SizePolicy> layout;

if (SizePolicy::Assign(layout.R1, R1))

{

size\_t const offsetOfRelativeJumpOffsetFromEnd = sizeof(OpLayoutT\_BrReg1<SizePolicy>) - offsetof(OpLayoutT\_BrReg1<SizePolicy>, RelativeJumpOffset);

layout.RelativeJumpOffset = offsetOfRelativeJumpOffsetFromEnd;

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

AddJumpOffset(op, labelID, offsetOfRelativeJumpOffsetFromEnd);

return true;

}

return false;

}

void ByteCodeWriter::BrReg1(OpCode op, ByteCodeLabel labelID, RegSlot R1)

{

CheckOpen();

CheckOp(op, OpLayoutType::BrReg1);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

CheckLabel(labelID);

R1 = ConsumeReg(R1);

MULTISIZE\_LAYOUT\_WRITE(BrReg1, op, labelID, R1);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteBrReg2(OpCode op, ByteCodeLabel labelID, RegSlot R1, RegSlot R2)

{

OpLayoutT\_BrReg2<SizePolicy> layout;

if (SizePolicy::Assign(layout.R1, R1) && SizePolicy::Assign(layout.R2, R2))

{

size\_t const offsetOfRelativeJumpOffsetFromEnd = sizeof(OpLayoutT\_BrReg2<SizePolicy>) - offsetof(OpLayoutT\_BrReg2<SizePolicy>, RelativeJumpOffset);

layout.RelativeJumpOffset = offsetOfRelativeJumpOffsetFromEnd;

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

AddJumpOffset(op, labelID, offsetOfRelativeJumpOffsetFromEnd);

return true;

}

return false;

}

void ByteCodeWriter::BrReg2(OpCode op, ByteCodeLabel labelID, RegSlot R1, RegSlot R2)

{

CheckOpen();

CheckOp(op, OpLayoutType::BrReg2);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

CheckLabel(labelID);

R1 = ConsumeReg(R1);

R2 = ConsumeReg(R2);

MULTISIZE\_LAYOUT\_WRITE(BrReg2, op, labelID, R1, R2);

}

void ByteCodeWriter::BrProperty(OpCode op, ByteCodeLabel labelID, RegSlot instance, PropertyIdIndexType index)

{

CheckOpen();

CheckOp(op, OpLayoutType::BrProperty);

Assert(!OpCodeAttr::HasMultiSizeLayout(op));

CheckLabel(labelID);

instance = ConsumeReg(instance);

size\_t const offsetOfRelativeJumpOffsetFromEnd = sizeof(OpLayoutBrProperty) - offsetof(OpLayoutBrProperty, RelativeJumpOffset);

OpLayoutBrProperty data;

data.RelativeJumpOffset = offsetOfRelativeJumpOffsetFromEnd;

data.Instance = instance;

data.PropertyIdIndex = index;

m\_byteCodeData.Encode(op, &data, sizeof(data), this);

AddJumpOffset(op, labelID, offsetOfRelativeJumpOffsetFromEnd);

}

void ByteCodeWriter::BrLocalProperty(OpCode op, ByteCodeLabel labelID, PropertyIdIndexType index)

{

CheckOpen();

CheckOp(op, OpLayoutType::BrLocalProperty);

Assert(!OpCodeAttr::HasMultiSizeLayout(op));

CheckLabel(labelID);

size\_t const offsetOfRelativeJumpOffsetFromEnd = sizeof(OpLayoutBrLocalProperty) - offsetof(OpLayoutBrLocalProperty, RelativeJumpOffset);

OpLayoutBrLocalProperty data;

data.RelativeJumpOffset = offsetOfRelativeJumpOffsetFromEnd;

data.PropertyIdIndex = index;

m\_byteCodeData.Encode(op, &data, sizeof(data), this);

AddJumpOffset(op, labelID, offsetOfRelativeJumpOffsetFromEnd);

}

void ByteCodeWriter::BrEnvProperty(OpCode op, ByteCodeLabel labelID, PropertyIdIndexType index, int32 slotIndex)

{

CheckOpen();

CheckOp(op, OpLayoutType::BrEnvProperty);

Assert(!OpCodeAttr::HasMultiSizeLayout(op));

CheckLabel(labelID);

size\_t const offsetOfRelativeJumpOffsetFromEnd = sizeof(OpLayoutBrEnvProperty) - offsetof(OpLayoutBrEnvProperty, RelativeJumpOffset);

OpLayoutBrEnvProperty data;

data.RelativeJumpOffset = offsetOfRelativeJumpOffsetFromEnd;

data.SlotIndex = slotIndex;

data.PropertyIdIndex = index;

m\_byteCodeData.Encode(op, &data, sizeof(data), this);

AddJumpOffset(op, labelID, offsetOfRelativeJumpOffsetFromEnd);

}

bool ByteCodeWriter::DoDynamicProfileOpcode(Phase tag, bool noHeuristics) const

{

#if ENABLE\_PROFILE\_INFO

if (!DynamicProfileInfo::IsEnabled(tag, this->m\_functionWrite))

{

return false;

}

// Other heuristics

switch (tag)

{

case Phase::InlinePhase:

// Do profile opcode everywhere if we are an inline candidate

// Otherwise, only in loops if the function has loop

#pragma prefast(suppress:6236, "DevDiv bug 830883. False positive when PHASE\_OFF is #defined as '(false)'.")

return PHASE\_FORCE(Phase::InlinePhase, this->m\_functionWrite) ||

(!this->m\_functionWrite->GetDontInline() &&

(noHeuristics || !this->m\_hasLoop || (this->m\_loopNest != 0) ||

!(PHASE\_OFF(InlineOutsideLoopsPhase, this->m\_functionWrite))));

default:

return true;

}

#else

return false;

#endif

}

bool ByteCodeWriter::ShouldIncrementCallSiteId(OpCode op)

{

if ((DoProfileCallOp(op) && DoDynamicProfileOpcode(InlinePhase)) ||

(DoProfileNewScObjArrayOp(op) && (DoDynamicProfileOpcode(NativeArrayPhase, true) || DoDynamicProfileOpcode(InlinePhase, true))) ||

(DoProfileNewScObjectOp(op) && (DoDynamicProfileOpcode(InlinePhase, true) || DoDynamicProfileOpcode(FixedNewObjPhase, true))))

{

return true;

}

return false;

}

void ByteCodeWriter::StartCall(OpCode op, ArgSlot ArgCount)

{

CheckOpen();

CheckOp(op, OpLayoutType::StartCall);

OpLayoutStartCall data;

data.ArgCount = ArgCount;

m\_byteCodeData.Encode(op, &data, sizeof(data), this);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteCallIExtended(OpCode op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, CallIExtendedOptions options, uint32 spreadArgsOffset)

{

OpLayoutT\_CallIExtended<SizePolicy> layout;

if (SizePolicy::Assign(layout.Return, returnValueRegister) && SizePolicy::Assign(layout.Function, functionRegister)

&& SizePolicy::Assign(layout.ArgCount, givenArgCount) && SizePolicy::Assign(layout.Options, options)

&& SizePolicy::Assign(layout.SpreadAuxOffset, spreadArgsOffset))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteCallIExtendedWithICIndex(OpCode op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, InlineCacheIndex inlineCacheIndex, bool isRootLoad, CallIExtendedOptions options, uint32 spreadArgsOffset)

{

OpLayoutT\_CallIExtendedWithICIndex<SizePolicy> layout;

if (SizePolicy::Assign(layout.Return, returnValueRegister) && SizePolicy::Assign(layout.Function, functionRegister)

&& SizePolicy::Assign(layout.ArgCount, givenArgCount) && SizePolicy::Assign(layout.inlineCacheIndex, inlineCacheIndex)

&& SizePolicy::Assign(layout.Options, options) && SizePolicy::Assign(layout.SpreadAuxOffset, spreadArgsOffset))

{

size\_t offset = m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

if (isRootLoad)

{

size\_t inlineCacheOffset = offset + OpCodeUtil::EncodedSize(op, SizePolicy::LayoutEnum)

+ offsetof(OpLayoutT\_CallIExtendedWithICIndex<SizePolicy>, inlineCacheIndex);

rootObjectLoadMethodInlineCacheOffsets.Prepend(m\_labelOffsets->GetAllocator(), inlineCacheOffset);

}

return true;

}

return false;

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteCallIExtendedFlags(OpCode op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, CallIExtendedOptions options, uint32 spreadArgsOffset, CallFlags callFlags)

{

OpLayoutT\_CallIExtendedFlags<SizePolicy> layout;

if (SizePolicy::Assign(layout.Return, returnValueRegister) && SizePolicy::Assign(layout.Function, functionRegister)

&& SizePolicy::Assign(layout.ArgCount, givenArgCount) && SizePolicy::Assign(layout.Options, options)

&& SizePolicy::Assign(layout.SpreadAuxOffset, spreadArgsOffset) && SizePolicy::Assign(layout.callFlags, callFlags))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteCallIExtendedFlagsWithICIndex(OpCode op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, InlineCacheIndex inlineCacheIndex, bool isRootLoad, CallIExtendedOptions options, uint32 spreadArgsOffset, CallFlags callFlags)

{

OpLayoutT\_CallIExtendedFlagsWithICIndex<SizePolicy> layout;

if (SizePolicy::Assign(layout.Return, returnValueRegister) && SizePolicy::Assign(layout.Function, functionRegister)

&& SizePolicy::Assign(layout.ArgCount, givenArgCount) && SizePolicy::Assign(layout.inlineCacheIndex, inlineCacheIndex)

&& SizePolicy::Assign(layout.Options, options) && SizePolicy::Assign(layout.SpreadAuxOffset, spreadArgsOffset)

&& SizePolicy::Assign(layout.callFlags, callFlags))

{

size\_t offset = m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

if (isRootLoad)

{

size\_t inlineCacheOffset = offset + OpCodeUtil::EncodedSize(op, SizePolicy::LayoutEnum)

+ offsetof(OpLayoutT\_CallIExtendedFlagsWithICIndex<SizePolicy>, inlineCacheIndex);

rootObjectLoadMethodInlineCacheOffsets.Prepend(m\_labelOffsets->GetAllocator(), inlineCacheOffset);

}

return true;

}

return false;

}

void ByteCodeWriter::CallIExtended(OpCode op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, CallIExtendedOptions options, const void \*buffer, uint byteCount, ProfileId callSiteId, CallFlags callFlags)

{

CheckOpen();

bool hasCallFlags = !(callFlags == CallFlags\_None);

if (hasCallFlags)

{

CheckOp(op, OpLayoutType::CallIExtendedFlags);

}

else

{

CheckOp(op, OpLayoutType::CallIExtended);

}

Assert(OpCodeAttr::HasMultiSizeLayout(op));

// givenArgCount could be <, ==, or > than Function's "InParams" count

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

{

returnValueRegister = ConsumeReg(returnValueRegister);

}

functionRegister = ConsumeReg(functionRegister);

// CallISpread is not going to use the ldFld cache index, but still remove it from the map as we expect

// the entry for a cache index to be removed once we have seen the corresponding call.

CacheIdUnit unit;

unit.cacheId = Js::Constants::NoInlineCacheIndex;

callRegToLdFldCacheIndexMap->TryGetValueAndRemove(functionRegister, &unit);

bool isProfiled = false, isProfiled2 = false;

ProfileId profileId = callSiteId, profileId2 = Constants::NoProfileId;

bool isCallWithICIndex = false;

if (DoProfileCallOp(op))

{

if (DoDynamicProfileOpcode(InlinePhase) &&

callSiteId != Js::Constants::NoProfileId)

{

op = Js::OpCodeUtil::ConvertCallOpToProfiled(op);

isProfiled = true;

}

else if ((DoDynamicProfileOpcode(AggressiveIntTypeSpecPhase) || DoDynamicProfileOpcode(FloatTypeSpecPhase)) &&

this->m\_functionWrite->AllocProfiledReturnTypeId(&profileId))

{

op = Js::OpCodeUtil::ConvertCallOpToProfiledReturnType(op);

isProfiled = true;

}

}

else if (DoProfileNewScObjArrayOp(op) &&

(DoDynamicProfileOpcode(NativeArrayPhase, true) || DoDynamicProfileOpcode(InlinePhase, true)) &&

callSiteId != Js::Constants::NoProfileId &&

this->m\_functionWrite->AllocProfiledArrayCallSiteId(&profileId2))

{

OpCodeUtil::ConvertNonCallOpToProfiled(op);

isProfiled = true;

isProfiled2 = true;

}

else if (DoProfileNewScObjectOp(op) && (DoDynamicProfileOpcode(InlinePhase, true) || DoDynamicProfileOpcode(FixedNewObjPhase, true)) &&

callSiteId != Js::Constants::NoProfileId)

{

OpCodeUtil::ConvertNonCallOpToProfiled(op);

isProfiled = true;

}

uint spreadArgsOffset = 0;

if (options & CallIExtended\_SpreadArgs)

{

Assert(buffer != nullptr && byteCount > 0);

spreadArgsOffset = InsertAuxiliaryData(buffer, byteCount);

}

if (isCallWithICIndex)

{

if (hasCallFlags == true)

{

MULTISIZE\_LAYOUT\_WRITE(CallIExtendedFlagsWithICIndex, op, returnValueRegister, functionRegister, givenArgCount, unit.cacheId, unit.isRootObjectCache, options, spreadArgsOffset, callFlags);

}

else

{

MULTISIZE\_LAYOUT\_WRITE(CallIExtendedWithICIndex, op, returnValueRegister, functionRegister, givenArgCount, unit.cacheId, unit.isRootObjectCache, options, spreadArgsOffset);

}

}

else

{

if (hasCallFlags == true)

{

MULTISIZE\_LAYOUT\_WRITE(CallIExtendedFlags, op, returnValueRegister, functionRegister, givenArgCount, options, spreadArgsOffset, callFlags);

}

else

{

MULTISIZE\_LAYOUT\_WRITE(CallIExtended, op, returnValueRegister, functionRegister, givenArgCount, options, spreadArgsOffset);

}

}

if (isProfiled)

{

m\_byteCodeData.Encode(&profileId, sizeof(Js::ProfileId));

if (isProfiled2)

{

m\_byteCodeData.Encode(&profileId2, sizeof(Js::ProfileId));

}

}

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteCallIWithICIndex(OpCode op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, InlineCacheIndex inlineCacheIndex, bool isRootLoad)

{

OpLayoutT\_CallIWithICIndex<SizePolicy> layout;

if (SizePolicy::Assign(layout.Return, returnValueRegister) && SizePolicy::Assign(layout.Function, functionRegister)

&& SizePolicy::Assign(layout.ArgCount, givenArgCount) && SizePolicy::Assign(layout.inlineCacheIndex, inlineCacheIndex))

{

size\_t offset = m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

if (isRootLoad)

{

size\_t inlineCacheOffset = offset + OpCodeUtil::EncodedSize(op, SizePolicy::LayoutEnum)

+ offsetof(OpLayoutT\_CallIWithICIndex<SizePolicy>, inlineCacheIndex);

rootObjectLoadMethodInlineCacheOffsets.Prepend(m\_labelOffsets->GetAllocator(), inlineCacheOffset);

}

return true;

}

return false;

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteCallIFlagsWithICIndex(OpCode op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, InlineCacheIndex inlineCacheIndex, bool isRootLoad, CallFlags callFlags)

{

OpLayoutT\_CallIFlagsWithICIndex<SizePolicy> layout;

if (SizePolicy::Assign(layout.Return, returnValueRegister) && SizePolicy::Assign(layout.Function, functionRegister)

&& SizePolicy::Assign(layout.ArgCount, givenArgCount) && SizePolicy::Assign(layout.inlineCacheIndex, inlineCacheIndex)

&& SizePolicy::Assign(layout.callFlags, callFlags))

{

size\_t offset = m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

if (isRootLoad)

{

size\_t inlineCacheOffset = offset + OpCodeUtil::EncodedSize(op, SizePolicy::LayoutEnum)

+ offsetof(OpLayoutT\_CallIFlagsWithICIndex<SizePolicy>, inlineCacheIndex);

rootObjectLoadMethodInlineCacheOffsets.Prepend(m\_labelOffsets->GetAllocator(), inlineCacheOffset);

}

return true;

}

return false;

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteCallI(OpCode op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount)

{

OpLayoutT\_CallI<SizePolicy> layout;

if (SizePolicy::Assign(layout.Return, returnValueRegister) && SizePolicy::Assign(layout.Function, functionRegister)

&& SizePolicy::Assign(layout.ArgCount, givenArgCount))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteCallIFlags(OpCode op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, CallFlags callFlags)

{

OpLayoutT\_CallIFlags<SizePolicy> layout;

if (SizePolicy::Assign(layout.Return, returnValueRegister) && SizePolicy::Assign(layout.Function, functionRegister)

&& SizePolicy::Assign(layout.ArgCount, givenArgCount) && SizePolicy::Assign(layout.callFlags, callFlags))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::RemoveEntryForRegSlotFromCacheIdMap(RegSlot regSlot)

{

regSlot = ConsumeReg(regSlot);

CacheIdUnit unit;

unit.cacheId = Js::Constants::NoInlineCacheIndex;

callRegToLdFldCacheIndexMap->TryGetValueAndRemove(regSlot, &unit);

}

void ByteCodeWriter::CallI(OpCode op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, ProfileId callSiteId, CallFlags callFlags)

{

CheckOpen();

bool hasCallFlags = !(callFlags == CallFlags\_None);

if (hasCallFlags == true)

{

CheckOp(op, OpLayoutType::CallIFlags);

}

else

{

CheckOp(op, OpLayoutType::CallI);

}

Assert(OpCodeAttr::HasMultiSizeLayout(op));

// givenArgCount could be <, ==, or > than Function's "InParams" count

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

{

returnValueRegister = ConsumeReg(returnValueRegister);

}

functionRegister = ConsumeReg(functionRegister);

bool isProfiled = false;

bool isProfiled2 = false;

bool isCallWithICIndex = false;

ProfileId profileId = callSiteId;

ProfileId profileId2 = Constants::NoProfileId;

CacheIdUnit unit;

unit.cacheId = Js::Constants::NoInlineCacheIndex;

callRegToLdFldCacheIndexMap->TryGetValueAndRemove(functionRegister, &unit);

if (DoProfileCallOp(op))

{

if (DoDynamicProfileOpcode(InlinePhase) &&

callSiteId != Js::Constants::NoProfileId)

{

if (unit.cacheId == Js::Constants::NoInlineCacheIndex)

{

op = Js::OpCodeUtil::ConvertCallOpToProfiled(op);

isProfiled = true;

}

else

{

isCallWithICIndex = true;

op = Js::OpCodeUtil::ConvertCallOpToProfiled(op, true);

isProfiled = true;

}

}

else if ((DoDynamicProfileOpcode(AggressiveIntTypeSpecPhase) || DoDynamicProfileOpcode(FloatTypeSpecPhase)) &&

this->m\_functionWrite->AllocProfiledReturnTypeId(&profileId))

{

op = Js::OpCodeUtil::ConvertCallOpToProfiledReturnType(op);

isProfiled = true;

}

}

else if (DoProfileNewScObjArrayOp(op) &&

(DoDynamicProfileOpcode(NativeArrayPhase, true) || DoDynamicProfileOpcode(InlinePhase, true)) &&

callSiteId != Js::Constants::NoProfileId &&

this->m\_functionWrite->AllocProfiledArrayCallSiteId(&profileId2))

{

OpCodeUtil::ConvertNonCallOpToProfiled(op);

isProfiled = true;

isProfiled2 = true;

}

else if (DoProfileNewScObjectOp(op) &&

(DoDynamicProfileOpcode(InlinePhase, true) || DoDynamicProfileOpcode(FixedNewObjPhase, true)) &&

callSiteId != Js::Constants::NoProfileId)

{

if (unit.cacheId == Js::Constants::NoInlineCacheIndex)

{

OpCodeUtil::ConvertNonCallOpToProfiled(op);

isProfiled = true;

}

else

{

isCallWithICIndex = true;

OpCodeUtil::ConvertNonCallOpToProfiledWithICIndex(op);

isProfiled = true;

}

}

if (isCallWithICIndex)

{

if (hasCallFlags == true)

{

MULTISIZE\_LAYOUT\_WRITE(CallIFlagsWithICIndex, op, returnValueRegister, functionRegister, givenArgCount, unit.cacheId, unit.isRootObjectCache, callFlags);

}

else

{

MULTISIZE\_LAYOUT\_WRITE(CallIWithICIndex, op, returnValueRegister, functionRegister, givenArgCount, unit.cacheId, unit.isRootObjectCache);

}

}

else

{

if (hasCallFlags == true)

{

MULTISIZE\_LAYOUT\_WRITE(CallIFlags, op, returnValueRegister, functionRegister, givenArgCount, callFlags);

}

else

{

MULTISIZE\_LAYOUT\_WRITE(CallI, op, returnValueRegister, functionRegister, givenArgCount);

}

}

if (isProfiled)

{

m\_byteCodeData.Encode(&profileId, sizeof(Js::ProfileId));

if (isProfiled2)

{

m\_byteCodeData.Encode(&profileId2, sizeof(Js::ProfileId));

}

}

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteElementI(OpCode op, RegSlot Value, RegSlot Instance, RegSlot Element)

{

OpLayoutT\_ElementI<SizePolicy> layout;

if (SizePolicy::Assign(layout.Value, Value) && SizePolicy::Assign(layout.Instance, Instance)

&& SizePolicy::Assign(layout.Element, Element))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::Element(OpCode op, RegSlot Value, RegSlot Instance, RegSlot Element, bool instanceAtReturnRegOK)

{

CheckOpen();

CheckOp(op, OpLayoutType::ElementI);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

Value = ConsumeReg(Value);

Instance = ConsumeReg(Instance);

Element = ConsumeReg(Element);

if (this->m\_functionWrite->GetIsStrictMode())

{

if (op == OpCode::DeleteElemI\_A)

{

op = OpCode::DeleteElemIStrict\_A;

}

}

bool isProfiledLayout = false;

Js::ProfileId profileId = Js::Constants::NoProfileId;

Assert(instanceAtReturnRegOK || Instance != 0);

if (DoDynamicProfileOpcode(AggressiveIntTypeSpecPhase) ||

DoDynamicProfileOpcode(FloatTypeSpecPhase) ||

DoDynamicProfileOpcode(TypedArrayTypeSpecPhase) ||

DoDynamicProfileOpcode(ArrayCheckHoistPhase))

{

OpCode newop;

switch (op)

{

case OpCode::LdElemI\_A:

newop = OpCode::ProfiledLdElemI\_A;

if (this->m\_functionWrite->AllocProfiledLdElemId(&profileId))

{

isProfiledLayout = true;

op = newop;

}

break;

case Js::OpCode::StElemI\_A:

newop = OpCode::ProfiledStElemI\_A;

goto StoreCommon;

case Js::OpCode::StElemI\_A\_Strict:

newop = OpCode::ProfiledStElemI\_A\_Strict;

StoreCommon:

if (this->m\_functionWrite->AllocProfiledStElemId(&profileId))

{

isProfiledLayout = true;

op = newop;

}

break;

}

}

MULTISIZE\_LAYOUT\_WRITE(ElementI, op, Value, Instance, Element);

if (isProfiledLayout)

{

Assert(profileId != Js::Constants::NoProfileId);

m\_byteCodeData.Encode(&profileId, sizeof(Js::ProfileId));

}

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteElementUnsigned1(OpCode op, RegSlot Value, RegSlot Instance, uint32 Element)

{

OpLayoutT\_ElementUnsigned1<SizePolicy> layout;

if (SizePolicy::Assign(layout.Value, Value) && SizePolicy::Assign(layout.Instance, Instance)

&& SizePolicy::Assign(layout.Element, Element))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::ElementUnsigned1(OpCode op, RegSlot Value, RegSlot Instance, uint32 Element)

{

CheckOpen();

CheckOp(op, OpLayoutType::ElementUnsigned1);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

Value = ConsumeReg(Value);

Instance = ConsumeReg(Instance);

MULTISIZE\_LAYOUT\_WRITE(ElementUnsigned1, op, Value, Instance, Element);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteElementScopedC(OpCode op, RegSlot value, PropertyIdIndexType propertyIdIndex)

{

OpLayoutT\_ElementScopedC<SizePolicy> layout;

if (SizePolicy::Assign(layout.Value, value)

&& SizePolicy::Assign(layout.PropertyIdIndex, propertyIdIndex))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::ScopedProperty(OpCode op, RegSlot value, PropertyIdIndexType propertyIdIndex)

{

CheckOpen();

CheckOp(op, OpLayoutType::ElementScopedC);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

value = ConsumeReg(value);

#if DBG

switch (op)

{

case OpCode::ScopedDeleteFld:

case OpCode::ScopedEnsureNoRedeclFld:

case OpCode::ScopedInitFunc:

break;

default:

AssertMsg(false, "The specified OpCode is not intended for scoped field-access");

break;

}

#endif

if (this->m\_functionWrite->GetIsStrictMode())

{

if (op == OpCode::ScopedDeleteFld)

{

op = OpCode::ScopedDeleteFldStrict;

}

}

MULTISIZE\_LAYOUT\_WRITE(ElementScopedC, op, value, propertyIdIndex);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteElementC(OpCode op, RegSlot value, RegSlot instance, PropertyIdIndexType propertyIdIndex)

{

OpLayoutT\_ElementC<SizePolicy> layout;

if (SizePolicy::Assign(layout.Value, value) && SizePolicy::Assign(layout.Instance, instance)

&& SizePolicy::Assign(layout.PropertyIdIndex, propertyIdIndex))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::Property(OpCode op, RegSlot value, RegSlot instance, PropertyIdIndexType propertyIdIndex)

{

CheckOpen();

CheckOp(op, OpLayoutType::ElementC);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

value = ConsumeReg(value);

instance = ConsumeReg(instance);

#if DBG

switch (op)

{

case OpCode::InitSetFld:

case OpCode::InitGetFld:

case OpCode::InitClassMemberGet:

case OpCode::InitClassMemberSet:

case OpCode::InitProto:

case OpCode::DeleteFld:

case OpCode::DeleteRootFld:

case OpCode::LdElemUndefScoped:

case OpCode::StFuncExpr:

break;

default:

AssertMsg(false, "The specified OpCode is not intended for field-access");

break;

}

#endif

if (this->m\_functionWrite->GetIsStrictMode())

{

if (op == OpCode::DeleteFld)

{

op = OpCode::DeleteFldStrict;

}

else if (op == OpCode::DeleteRootFld)

{

// We will reach here when in the language service mode, since in that mode we have skipped that error.

op = OpCode::DeleteRootFldStrict;

}

}

MULTISIZE\_LAYOUT\_WRITE(ElementC, op, value, instance, propertyIdIndex);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteElementSlot(OpCode op, RegSlot value, RegSlot instance, int32 slotId)

{

OpLayoutT\_ElementSlot<SizePolicy> layout;

if (SizePolicy::Assign(layout.Value, value) && SizePolicy::Assign(layout.Instance, instance)

&& SizePolicy::Assign(layout.SlotIndex, slotId))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::Slot(OpCode op, RegSlot value, RegSlot instance, int32 slotId)

{

CheckOpen();

CheckOp(op, OpLayoutType::ElementSlot);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

value = ConsumeReg(value);

instance = ConsumeReg(instance);

#if DBG

switch (op)

{

#if ENABLE\_NATIVE\_CODEGEN

case OpCode::LdSlotArr:

case OpCode::StSlot:

case OpCode::StSlotChkUndecl:

#endif

case OpCode::StObjSlot:

case OpCode::StObjSlotChkUndecl:

break;

default:

AssertMsg(false, "The specified OpCode is not intended for slot access");

break;

}

#endif

MULTISIZE\_LAYOUT\_WRITE(ElementSlot, op, value, instance, slotId);

}

void ByteCodeWriter::Slot(OpCode op, RegSlot value, RegSlot instance, int32 slotId, ProfileId profileId)

{

CheckOpen();

CheckOp(op, OpLayoutType::ElementSlot);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

value = ConsumeReg(value);

instance = ConsumeReg(instance);

switch (op)

{

case OpCode::LdSlot:

case OpCode::LdObjSlot:

if ((DoDynamicProfileOpcode(AggressiveIntTypeSpecPhase) || DoDynamicProfileOpcode(FloatTypeSpecPhase)) &&

profileId != Constants::NoProfileId)

{

OpCodeUtil::ConvertNonCallOpToProfiled(op);

}

break;

default:

AssertMsg(false, "The specified OpCode is not intended for slot access");

break;

}

MULTISIZE\_LAYOUT\_WRITE(ElementSlot, op, value, instance, slotId);

if (OpCodeAttr::IsProfiledOp(op))

{

m\_byteCodeData.Encode(&profileId, sizeof(Js::ProfileId));

}

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteElementSlotI1(OpCode op, RegSlot value, int32 slotId)

{

OpLayoutT\_ElementSlotI1<SizePolicy> layout;

if (SizePolicy::Assign(layout.Value, value)

&& SizePolicy::Assign(layout.SlotIndex, slotId))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::SlotI1(OpCode op, RegSlot value, int32 slotId)

{

CheckOpen();

CheckOp(op, OpLayoutType::ElementSlotI1);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

value = ConsumeReg(value);

#if DBG

switch (op)

{

case OpCode::LdEnvObj:

case OpCode::StLocalSlot:

case OpCode::StLocalObjSlot:

case OpCode::StLocalSlotChkUndecl:

case OpCode::StLocalObjSlotChkUndecl:

{

break;

}

default:

{

AssertMsg(false, "The specified OpCode is not intended for slot access");

break;

}

}

#endif

MULTISIZE\_LAYOUT\_WRITE(ElementSlotI1, op, value, slotId);

}

void ByteCodeWriter::SlotI1(OpCode op, RegSlot value, int32 slotId, ProfileId profileId)

{

CheckOpen();

CheckOp(op, OpLayoutType::ElementSlotI1);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

value = ConsumeReg(value);

switch (op)

{

case OpCode::LdLocalSlot:

case OpCode::LdLocalObjSlot:

if ((DoDynamicProfileOpcode(AggressiveIntTypeSpecPhase) || DoDynamicProfileOpcode(FloatTypeSpecPhase)) &&

profileId != Constants::NoProfileId)

{

OpCodeUtil::ConvertNonCallOpToProfiled(op);

}

break;

default:

{

AssertMsg(false, "The specified OpCode is not intended for slot access");

break;

}

}

MULTISIZE\_LAYOUT\_WRITE(ElementSlotI1, op, value, slotId);

if (OpCodeAttr::IsProfiledOp(op))

{

m\_byteCodeData.Encode(&profileId, sizeof(Js::ProfileId));

}

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteElementSlotI2(OpCode op, RegSlot value, int32 slotId1, int32 slotId2)

{

OpLayoutT\_ElementSlotI2<SizePolicy> layout;

if (SizePolicy::Assign(layout.Value, value)

&& SizePolicy::Assign(layout.SlotIndex1, slotId1)

&& SizePolicy::Assign(layout.SlotIndex2, slotId2))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::SlotI2(OpCode op, RegSlot value, int32 slotId1, int32 slotId2)

{

CheckOpen();

CheckOp(op, OpLayoutType::ElementSlotI2);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

value = ConsumeReg(value);

#if DBG

switch (op)

{

case OpCode::StInnerSlot:

case OpCode::StInnerSlotChkUndecl:

case OpCode::StInnerObjSlot:

case OpCode::StInnerObjSlotChkUndecl:

case OpCode::StEnvSlot:

case OpCode::StEnvSlotChkUndecl:

case OpCode::StEnvObjSlot:

case OpCode::StEnvObjSlotChkUndecl:

{

break;

}

default:

{

AssertMsg(false, "The specified OpCode is not intended for slot access");

break;

}

}

#endif

MULTISIZE\_LAYOUT\_WRITE(ElementSlotI2, op, value, slotId1, slotId2);

}

void ByteCodeWriter::SlotI2(OpCode op, RegSlot value, int32 slotId1, int32 slotId2, ProfileId profileId)

{

CheckOpen();

CheckOp(op, OpLayoutType::ElementSlotI2);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

value = ConsumeReg(value);

switch (op)

{

case OpCode::LdInnerSlot:

case OpCode::LdInnerObjSlot:

case OpCode::LdEnvSlot:

case OpCode::LdEnvObjSlot:

if ((DoDynamicProfileOpcode(AggressiveIntTypeSpecPhase) || DoDynamicProfileOpcode(FloatTypeSpecPhase)) &&

profileId != Constants::NoProfileId)

{

OpCodeUtil::ConvertNonCallOpToProfiled(op);

}

break;

default:

{

AssertMsg(false, "The specified OpCode is not intended for slot access");

break;

}

}

MULTISIZE\_LAYOUT\_WRITE(ElementSlotI2, op, value, slotId1, slotId2);

if (OpCodeAttr::IsProfiledOp(op))

{

m\_byteCodeData.Encode(&profileId, sizeof(Js::ProfileId));

}

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteElementU(OpCode op, RegSlot instance, PropertyIdIndexType index)

{

OpLayoutT\_ElementU<SizePolicy> layout;

if (SizePolicy::Assign(layout.Instance, instance) && SizePolicy::Assign(layout.PropertyIdIndex, index))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::ElementU(OpCode op, RegSlot instance, PropertyIdIndexType index)

{

CheckOpen();

CheckOp(op, OpLayoutType::ElementU);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

instance = ConsumeReg(instance);

MULTISIZE\_LAYOUT\_WRITE(ElementU, op, instance, index);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteElementScopedU(OpCode op, PropertyIdIndexType index)

{

OpLayoutT\_ElementScopedU<SizePolicy> layout;

if (SizePolicy::Assign(layout.PropertyIdIndex, index))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::ElementScopedU(OpCode op, PropertyIdIndexType index)

{

CheckOpen();

CheckOp(op, OpLayoutType::ElementScopedU);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

MULTISIZE\_LAYOUT\_WRITE(ElementScopedU, op, index);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteElementRootU(OpCode op, PropertyIdIndexType index)

{

OpLayoutT\_ElementRootU<SizePolicy> layout;

if (SizePolicy::Assign(layout.PropertyIdIndex, index))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::ElementRootU(OpCode op, PropertyIdIndexType index)

{

CheckOpen();

CheckOp(op, OpLayoutType::ElementRootU);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

MULTISIZE\_LAYOUT\_WRITE(ElementRootU, op, index);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteElementRootCP(OpCode op, RegSlot value, uint cacheId, bool isLoadMethod, bool isStore)

{

Assert(!isLoadMethod || !isStore);

OpLayoutT\_ElementRootCP<SizePolicy> layout;

if (SizePolicy::Assign(layout.Value, value) && SizePolicy::Assign(layout.inlineCacheIndex, cacheId))

{

size\_t offset = m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

size\_t inlineCacheOffset = offset + OpCodeUtil::EncodedSize(op, SizePolicy::LayoutEnum)

+ offsetof(OpLayoutT\_ElementRootCP<SizePolicy>, inlineCacheIndex);

// Root object inline cache index are given out from 0, but it will be at index after

// all the plain inline cache. Store the offset of the inline cache index to patch it up later.

SListBase<size\_t> \* rootObjectInlineCacheOffsets = isStore ?

&rootObjectStoreInlineCacheOffsets : isLoadMethod ? &rootObjectLoadMethodInlineCacheOffsets : &rootObjectLoadInlineCacheOffsets;

rootObjectInlineCacheOffsets->Prepend(this->m\_labelOffsets->GetAllocator(), inlineCacheOffset);

return true;

}

return false;

}

void ByteCodeWriter::PatchableRootProperty(OpCode op, RegSlot value, uint cacheId, bool isLoadMethod, bool isStore, bool registerCacheIdForCall)

{

CheckOpen();

CheckOp(op, OpLayoutType::ElementRootCP);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

Assert(!isLoadMethod || !isStore);

value = ConsumeReg(value);

switch (op)

{

case OpCode::LdRootFld:

case OpCode::LdRootFldForTypeOf:

if (DoDynamicProfileOpcode(AggressiveIntTypeSpecPhase) ||

DoDynamicProfileOpcode(FloatTypeSpecPhase) ||

DoDynamicProfileOpcode(ObjTypeSpecPhase) ||

DoDynamicProfileOpcode(InlinePhase) ||

DoDynamicProfileOpcode(ProfileBasedFldFastPathPhase))

{

OpCodeUtil::ConvertNonCallOpToProfiled(op);

}

break;

case OpCode::LdRootMethodFld:

if (registerCacheIdForCall)

{

CacheIdUnit unit(cacheId, true);

Assert(!callRegToLdFldCacheIndexMap->TryGetValue(value, &unit));

callRegToLdFldCacheIndexMap->Add(value, unit);

}

case OpCode::StRootFld:

case OpCode::StRootFldStrict:

case OpCode::InitRootFld:

if (DoDynamicProfileOpcode(ProfileBasedFldFastPathPhase) ||

DoDynamicProfileOpcode(InlinePhase) ||

DoDynamicProfileOpcode(ObjTypeSpecPhase))

{

OpCodeUtil::ConvertNonCallOpToProfiled(op);

}

break;

case OpCode::InitRootLetFld:

case OpCode::InitRootConstFld:

break;

default:

AssertMsg(false, "The specified OpCode is not intended for patchable root field-access");

break;

}

MULTISIZE\_LAYOUT\_WRITE(ElementRootCP, op, value, cacheId, isLoadMethod, isStore);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteElementP(OpCode op, RegSlot value, CacheId cacheId)

{

OpLayoutT\_ElementP<SizePolicy> layout;

if (SizePolicy::Assign(layout.Value, value)

&& SizePolicy::Assign(layout.inlineCacheIndex, cacheId))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::ElementP(OpCode op, RegSlot value, uint cacheId, bool isCtor, bool registerCacheIdForCall)

{

CheckOpen();

CheckOp(op, OpLayoutType::ElementP);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

value = ConsumeReg(value);

switch (op)

{

case OpCode::ScopedLdFld:

case OpCode::ScopedLdFldForTypeOf:

case OpCode::ScopedStFld:

case OpCode::ConsoleScopedStFld:

case OpCode::ScopedStFldStrict:

break;

case OpCode::LdLocalFld:

if (isCtor) // The symbol loaded by this LdFld will be used as a constructor

{

if (registerCacheIdForCall)

{

CacheIdUnit unit(cacheId);

Assert(!callRegToLdFldCacheIndexMap->TryGetValue(value, &unit));

callRegToLdFldCacheIndexMap->Add(value, unit);

}

}

if (DoDynamicProfileOpcode(AggressiveIntTypeSpecPhase) ||

DoDynamicProfileOpcode(FloatTypeSpecPhase) ||

DoDynamicProfileOpcode(ObjTypeSpecPhase) ||

DoDynamicProfileOpcode(InlinePhase) ||

DoDynamicProfileOpcode(ProfileBasedFldFastPathPhase))

{

OpCodeUtil::ConvertNonCallOpToProfiled(op);

}

break;

case OpCode::LdLocalMethodFld:

if (registerCacheIdForCall)

{

CacheIdUnit unit(cacheId);

Assert(!callRegToLdFldCacheIndexMap->TryGetValue(value, &unit));

callRegToLdFldCacheIndexMap->Add(value, unit);

}

// fall-through

case OpCode::StLocalFld:

case OpCode::InitLocalFld:

if (DoDynamicProfileOpcode(ProfileBasedFldFastPathPhase) ||

DoDynamicProfileOpcode(InlinePhase) ||

DoDynamicProfileOpcode(ObjTypeSpecPhase))

{

OpCodeUtil::ConvertNonCallOpToProfiled(op);

}

break;

case OpCode::InitLocalLetFld:

case OpCode::InitUndeclLocalLetFld:

case OpCode::InitUndeclLocalConstFld:

break;

default:

AssertMsg(false, "The specified OpCode not intended for base-less patchable field access");

break;

}

MULTISIZE\_LAYOUT\_WRITE(ElementP, op, value, cacheId);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteElementPIndexed(OpCode op, RegSlot value, uint32 scopeIndex, CacheId cacheId)

{

OpLayoutT\_ElementPIndexed<SizePolicy> layout;

if (SizePolicy::Assign(layout.Value, value)

&& SizePolicy::Assign(layout.inlineCacheIndex, cacheId)

&& SizePolicy::Assign(layout.scopeIndex, scopeIndex))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::ElementPIndexed(OpCode op, RegSlot value, uint32 scopeIndex, uint cacheId)

{

CheckOpen();

CheckOp(op, OpLayoutType::ElementPIndexed);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

value = ConsumeReg(value);

switch (op)

{

case OpCode::InitInnerFld:

case OpCode::InitInnerLetFld:

case OpCode::InitUndeclLetFld:

case OpCode::InitUndeclConstFld:

break;

break;

default:

AssertMsg(false, "The specified OpCode not intended for base-less patchable inner field access");

break;

}

MULTISIZE\_LAYOUT\_WRITE(ElementPIndexed, op, value, scopeIndex, cacheId);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteElementCP(OpCode op, RegSlot value, RegSlot instance, CacheId cacheId)

{

OpLayoutT\_ElementCP<SizePolicy> layout;

if (SizePolicy::Assign(layout.Value, value) && SizePolicy::Assign(layout.Instance, instance)

&& SizePolicy::Assign(layout.inlineCacheIndex, cacheId))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::PatchableProperty(OpCode op, RegSlot value, RegSlot instance, uint cacheId, bool isCtor, bool registerCacheIdForCall)

{

CheckOpen();

CheckOp(op, OpLayoutType::ElementCP);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

value = ConsumeReg(value);

instance = ConsumeReg(instance);

switch (op)

{

case OpCode::LdFldForTypeOf:

case OpCode::LdFld:

if (isCtor) // The symbol loaded by this LdFld will be used as a constructor

{

if (registerCacheIdForCall)

{

CacheIdUnit unit(cacheId);

Assert(!callRegToLdFldCacheIndexMap->TryGetValue(value, &unit));

callRegToLdFldCacheIndexMap->Add(value, unit);

}

}

case OpCode::LdFldForCallApplyTarget:

if (DoDynamicProfileOpcode(AggressiveIntTypeSpecPhase) ||

DoDynamicProfileOpcode(FloatTypeSpecPhase) ||

DoDynamicProfileOpcode(ObjTypeSpecPhase) ||

DoDynamicProfileOpcode(InlinePhase) ||

DoDynamicProfileOpcode(ProfileBasedFldFastPathPhase))

{

OpCodeUtil::ConvertNonCallOpToProfiled(op);

}

break;

case OpCode::LdMethodFld:

if (registerCacheIdForCall)

{

CacheIdUnit unit(cacheId);

Assert(!callRegToLdFldCacheIndexMap->TryGetValue(value, &unit));

callRegToLdFldCacheIndexMap->Add(value, unit);

}

// fall-through

case OpCode::StFld:

case OpCode::StFldStrict:

case OpCode::InitFld:

if (DoDynamicProfileOpcode(ProfileBasedFldFastPathPhase) ||

DoDynamicProfileOpcode(InlinePhase) ||

DoDynamicProfileOpcode(ObjTypeSpecPhase))

{

OpCodeUtil::ConvertNonCallOpToProfiled(op);

}

break;

case OpCode::InitLetFld:

case OpCode::InitConstFld:

case OpCode::InitClassMember:

case OpCode::ScopedLdMethodFld:

break;

default:

AssertMsg(false, "The specified OpCode is not intended for patchable field-access");

break;

}

MULTISIZE\_LAYOUT\_WRITE(ElementCP, op, value, instance, cacheId);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteElementC2(OpCode op, RegSlot value, RegSlot instance, PropertyIdIndexType propertyIdIndex, RegSlot value2)

{

OpLayoutT\_ElementC2<SizePolicy> layout;

if (SizePolicy::Assign(layout.Value, value) && SizePolicy::Assign(layout.Instance, instance)

&& SizePolicy::Assign(layout.PropertyIdIndex, propertyIdIndex) && SizePolicy::Assign(layout.Value2, value2))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::PatchablePropertyWithThisPtr(OpCode op, RegSlot value, RegSlot instance, RegSlot thisInstance, uint cacheId, bool isCtor, bool registerCacheIdForCall)

{

CheckOpen();

CheckOp(op, OpLayoutType::ElementC2);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

value = ConsumeReg(value);

instance = ConsumeReg(instance);

thisInstance = ConsumeReg(thisInstance);

switch (op)

{

case OpCode::LdSuperFld:

if (isCtor) // The symbol loaded by this LdSuperFld will be used as a constructor

{

if (registerCacheIdForCall)

{

CacheIdUnit unit(cacheId);

Assert(!callRegToLdFldCacheIndexMap->TryGetValue(value, &unit));

callRegToLdFldCacheIndexMap->Add(value, unit);

}

}

if (DoDynamicProfileOpcode(AggressiveIntTypeSpecPhase) ||

DoDynamicProfileOpcode(FloatTypeSpecPhase) ||

DoDynamicProfileOpcode(ObjTypeSpecPhase) ||

DoDynamicProfileOpcode(InlinePhase) ||

DoDynamicProfileOpcode(ProfileBasedFldFastPathPhase))

{

OpCodeUtil::ConvertNonCallOpToProfiled(op);

}

break;

case OpCode::StSuperFld:

if (DoDynamicProfileOpcode(ProfileBasedFldFastPathPhase) ||

DoDynamicProfileOpcode(InlinePhase) ||

DoDynamicProfileOpcode(ObjTypeSpecPhase))

{

OpCodeUtil::ConvertNonCallOpToProfiled(op);

}

break;

default:

AssertMsg(false, "The specified OpCode is not intended for patchable super field-access");

break;

}

MULTISIZE\_LAYOUT\_WRITE(ElementC2, op, value, instance, cacheId, thisInstance);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteElementScopedC2(OpCode op, RegSlot value, PropertyIdIndexType propertyIdIndex, RegSlot value2)

{

OpLayoutT\_ElementScopedC2<SizePolicy> layout;

if (SizePolicy::Assign(layout.Value, value)

&& SizePolicy::Assign(layout.PropertyIdIndex, propertyIdIndex) && SizePolicy::Assign(layout.Value2, value2))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::ScopedProperty2(OpCode op, RegSlot value, PropertyIdIndexType propertyIdIndex, RegSlot value2)

{

CheckOpen();

CheckOp(op, OpLayoutType::ElementScopedC2);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

value = ConsumeReg(value);

value2 = ConsumeReg(value2);

switch (op)

{

case OpCode::ScopedLdInst:

break;

default:

AssertMsg(false, "The specified OpCode is not intended for field-access with a second instance");

break;

}

MULTISIZE\_LAYOUT\_WRITE(ElementScopedC2, op, value, propertyIdIndex, value2);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteClass(OpCode op, RegSlot constructor, RegSlot extends)

{

OpLayoutT\_Class<SizePolicy> layout;

if (SizePolicy::Assign(layout.Constructor, constructor) && SizePolicy::Assign(layout.Extends, extends))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::InitClass(RegSlot constructor, RegSlot extends)

{

Assert(OpCodeAttr::HasMultiSizeLayout(Js::OpCode::InitClass));

CheckOpen();

constructor = ConsumeReg(constructor);

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

{

extends = ConsumeReg(extends);

}

MULTISIZE\_LAYOUT\_WRITE(Class, Js::OpCode::InitClass, constructor, extends);

}

void ByteCodeWriter::NewFunction(RegSlot destinationRegister, uint index, bool isGenerator)

{

CheckOpen();

destinationRegister = ConsumeReg(destinationRegister);

OpCode opcode = isGenerator ?

OpCode::NewScGenFunc :

this->m\_functionWrite->DoStackNestedFunc() ?

OpCode::NewStackScFunc : OpCode::NewScFunc;

Assert(OpCodeAttr::HasMultiSizeLayout(opcode));

MULTISIZE\_LAYOUT\_WRITE(ElementSlotI1, opcode, destinationRegister, index);

}

void ByteCodeWriter::NewInnerFunction(RegSlot destinationRegister, uint index, RegSlot environmentRegister, bool isGenerator)

{

CheckOpen();

destinationRegister = ConsumeReg(destinationRegister);

environmentRegister = ConsumeReg(environmentRegister);

OpCode opcode = isGenerator ?

OpCode::NewInnerScGenFunc :

this->m\_functionWrite->DoStackNestedFunc() ?

OpCode::NewInnerStackScFunc : OpCode::NewInnerScFunc;

Assert(OpCodeAttr::HasMultiSizeLayout(opcode));

MULTISIZE\_LAYOUT\_WRITE(ElementSlot, opcode, destinationRegister, environmentRegister, index);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteReg1Unsigned1(OpCode op, RegSlot R0, uint C1)

{

OpLayoutT\_Reg1Unsigned1<SizePolicy> layout;

if (SizePolicy::Assign(layout.R0, R0) && SizePolicy::Assign(layout.C1, C1))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::Reg1Unsigned1(OpCode op, RegSlot R0, uint C1)

{

CheckOpen();

CheckOp(op, OpLayoutType::Reg1Unsigned1);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

R0 = ConsumeReg(R0);

ProfileId profileId = Constants::NoProfileId;

bool isProfiled = DoProfileNewScArrayOp(op) &&

DoDynamicProfileOpcode(NativeArrayPhase, true) &&

this->m\_functionWrite->AllocProfiledArrayCallSiteId(&profileId);

if (isProfiled)

{

OpCodeUtil::ConvertNonCallOpToProfiled(op);

}

MULTISIZE\_LAYOUT\_WRITE(Reg1Unsigned1, op, R0, C1);

if (isProfiled)

{

m\_byteCodeData.Encode(&profileId, sizeof(Js::ProfileId));

}

}

void ByteCodeWriter::W1(OpCode op, ushort C1)

{

CheckOpen();

CheckOp(op, OpLayoutType::W1);

Assert(!OpCodeAttr::HasMultiSizeLayout(op));

OpLayoutW1 data;

data.C1 = C1;

m\_byteCodeData.Encode(op, &data, sizeof(data), this);

}

void ByteCodeWriter::Reg1Int2(OpCode op, RegSlot R0, int C1, int C2)

{

CheckOpen();

CheckOp(op, OpLayoutType::Reg1Int2);

Assert(!OpCodeAttr::HasMultiSizeLayout(op));

R0 = ConsumeReg(R0);

OpLayoutReg1Int2 data;

data.R0 = R0;

data.C1 = C1;

data.C2 = C2;

m\_byteCodeData.Encode(op, &data, sizeof(data), this);

}

template <typename SizePolicy>

bool ByteCodeWriter::TryWriteReg2Int1(OpCode op, RegSlot R0, RegSlot R1, int C1)

{

OpLayoutT\_Reg2Int1<SizePolicy> layout;

if (SizePolicy::Assign(layout.R0, R0) && SizePolicy::Assign(layout.R1, R1) && SizePolicy::Assign(layout.C1, C1))

{

m\_byteCodeData.EncodeT<SizePolicy::LayoutEnum>(op, &layout, sizeof(layout), this);

return true;

}

return false;

}

void ByteCodeWriter::Reg2Int1(OpCode op, RegSlot R0, RegSlot R1, int C1)

{

CheckOpen();

CheckOp(op, OpLayoutType::Reg2Int1);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

if (DoDynamicProfileOpcode(CheckThisPhase) ||

DoDynamicProfileOpcode(TypedArrayTypeSpecPhase) ||

DoDynamicProfileOpcode(ArrayCheckHoistPhase))

{

if (op == OpCode::LdThis)

{

op = OpCode::ProfiledLdThis;

}

}

R0 = ConsumeReg(R0);

R1 = ConsumeReg(R1);

MULTISIZE\_LAYOUT\_WRITE(Reg2Int1, op, R0, R1, C1);

}

void ByteCodeWriter::Num3(OpCode op, RegSlot C0, RegSlot C1, RegSlot C2)

{

CheckOpen();

CheckOp(op, OpLayoutType::Reg3);

Assert(OpCodeAttr::HasMultiSizeLayout(op));

MULTISIZE\_LAYOUT\_WRITE(Reg3, op, C0, C1, C2);

}

int ByteCodeWriter::AuxNoReg(OpCode op, const void\* buffer, int byteCount, int C1)

{

CheckOpen();

//

// Write the buffer's contents

//

int currentOffset = InsertAuxiliaryData(buffer, byteCount);

//

// Write OpCode to create new auxiliary data

//

OpLayoutAuxNoReg data;

data.Offset = currentOffset;

data.C1 = C1;

m\_byteCodeData.Encode(op, &data, sizeof(data), this);

return currentOffset;

}

void ByteCodeWriter::AuxNoReg(OpCode op, uint byteOffset, int C1)

{

CheckOpen();

//

// Write the buffer's contents

//

Assert(byteOffset < m\_auxiliaryData.GetCurrentOffset());

OpLayoutAuxNoReg data;

data.Offset = byteOffset;

data.C1 = C1;

m\_byteCodeData.Encode(op, &data, sizeof(data), this);

}

int ByteCodeWriter::Auxiliary(OpCode op, RegSlot destinationRegister, const void\* buffer, int byteCount, int C1)

{

CheckOpen();

destinationRegister = ConsumeReg(destinationRegister);

//

// Write the buffer's contents

//

int currentOffset = InsertAuxiliaryData(buffer, byteCount);

//

// Write OpCode to create new auxiliary data

//

ProfileId profileId = Constants::NoProfileId;

if (DoProfileNewScArrayOp(op) &&

DoDynamicProfileOpcode(NativeArrayPhase, true) &&

this->m\_functionWrite->AllocProfiledArrayCallSiteId(&profileId))

{

OpCodeUtil::ConvertNonCallOpToProfiled(op);

OpLayoutDynamicProfile<OpLayoutAuxiliary> data;

data.R0 = destinationRegister;

data.Offset = currentOffset;

data.C1 = C1;

data.profileId = profileId;

m\_byteCodeData.Encode(op, &data, sizeof(data), this);

}

else

{

OpLayoutAuxiliary data;

data.R0 = destinationRegister;

data.Offset = currentOffset;

data.C1 = C1;

m\_byteCodeData.Encode(op, &data, sizeof(data), this);

}

return currentOffset;

}

void ByteCodeWriter::Auxiliary(OpCode op, RegSlot destinationRegister, uint byteOffset, int C1)

{

CheckOpen();

destinationRegister = ConsumeReg(destinationRegister);

//

// Write the buffer's contents

//

Assert(byteOffset < m\_auxiliaryData.GetCurrentOffset());

OpLayoutAuxiliary data;

data.R0 = destinationRegister;

data.Offset = byteOffset;

data.C1 = C1;

m\_byteCodeData.Encode(op, &data, sizeof(data), this);

}

int ByteCodeWriter::Reg2Aux(OpCode op, RegSlot R0, RegSlot R1, const void\* buffer, int byteCount, int C1)

{

CheckOpen();

R0 = ConsumeReg(R0);

R1 = ConsumeReg(R1);

//

// Write the buffer's contents

//

int currentOffset = InsertAuxiliaryData(buffer, byteCount);

//

// Write OpCode to create new auxiliary data

//

OpLayoutReg2Aux data;

data.R0 = R0;

data.R1 = R1;

data.Offset = currentOffset;

data.C1 = C1;

m\_byteCodeData.Encode(op, &data, sizeof(data), this);

return currentOffset;

}

void ByteCodeWriter::Reg2Aux(OpCode op, RegSlot R0, RegSlot R1, uint byteOffset, int C1)

{

CheckOpen();

R0 = ConsumeReg(R0);

R1 = ConsumeReg(R1);

//

// Write the buffer's contents

//

Assert(byteOffset < m\_auxiliaryData.GetCurrentOffset());

OpLayoutReg2Aux data;

data.R0 = R0;

data.R1 = R1;

data.Offset = byteOffset;

data.C1 = C1;

m\_byteCodeData.Encode(op, &data, sizeof(data), this);

}

void ByteCodeWriter::AuxiliaryContext(OpCode op, RegSlot destinationRegister, const void\* buffer, int byteCount, Js::RegSlot C1)

{

CheckOpen();

destinationRegister = ConsumeReg(destinationRegister);

C1 = ConsumeReg(C1);

//

// Write the buffer's contents

//

int currentOffset = m\_auxContextData.GetCurrentOffset();

if (byteCount > 0)

{

m\_auxContextData.Encode(buffer, byteCount);

}

//

// Write OpCode to create new auxiliary data

//

OpLayoutAuxiliary data;

data.R0 = destinationRegister;

data.Offset = currentOffset;

data.C1 = C1;

m\_byteCodeData.Encode(op, &data, sizeof(data), this);

}

uint ByteCodeWriter::InsertAuxiliaryData(const void\* buffer, uint byteCount)

{

uint offset = m\_auxiliaryData.GetCurrentOffset();

if (byteCount > 0)

{

m\_auxiliaryData.Encode(buffer, byteCount);

}

return offset;

}

ByteCodeLabel ByteCodeWriter::DefineLabel()

{

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

if (m\_labelOffsets->Count() == INT\_MAX)

{

// Reach our limit

Js::Throw::OutOfMemory();

}

#else

// 32-bit machine don't have enough address space to get to INT\_MAX

Assert(m\_labelOffsets->Count() < INT\_MAX);

#endif

//

// Allocate a new label:

// - All label locations start as "undefined: -1". Once the label's location is marked in

// the byte-code, this will be updated.

//

return (ByteCodeLabel)m\_labelOffsets->Add(UINT\_MAX);

}

void ByteCodeWriter::MarkLabel(ByteCodeLabel labelID)

{

CheckOpen();

CheckLabel(labelID);

#ifdef BYTECODE\_BRANCH\_ISLAND

if (useBranchIsland)

{

// If we are going to emit a branch island, it should be before the label.

EnsureLongBranch(Js::OpCode::Label);

}

#endif

//

// Define the label as the current offset within the byte-code.

//

AssertMsg(m\_labelOffsets->Item(labelID) == UINT\_MAX, "A label may only be defined at one location");

m\_labelOffsets->SetExistingItem(labelID, m\_byteCodeData.GetCurrentOffset());

}

void ByteCodeWriter::AddJumpOffset(Js::OpCode op, ByteCodeLabel labelId, uint fieldByteOffsetFromEnd) // Offset of "Offset" field in OpLayout, in bytes

{

AssertMsg(fieldByteOffsetFromEnd < 100, "Ensure valid field offset");

CheckOpen();

CheckLabel(labelId);

uint jumpByteOffset = m\_byteCodeData.GetCurrentOffset() - fieldByteOffsetFromEnd;

#ifdef BYTECODE\_BRANCH\_ISLAND

if (useBranchIsland)

{

// Any Jump might need a long jump, account for that emit the branch island earlier.

// Even if it is a back edge and we are going to emit a long jump, we will still

// emit a branch around any way.

this->nextBranchIslandOffset -= LongBranchSize;

uint labelOffset = m\_labelOffsets->Item(labelId);

if (labelOffset != UINT\_MAX)

{

// Back branch, see if it needs to be long

Assert(labelOffset < m\_byteCodeData.GetCurrentOffset());

LongJumpOffset jumpOffset = labelOffset - m\_byteCodeData.GetCurrentOffset();

if (jumpOffset < -GetBranchLimit())

{

// Create the long jump label and add the original jump offset to the list first

ByteCodeLabel longJumpLabel = this->DefineLabel();

JumpInfo jumpInfo = { longJumpLabel, jumpByteOffset };

m\_jumpOffsets->Add(jumpInfo);

// Emit the jump around (if necessary)

ByteCodeLabel jumpAroundLabel = (ByteCodeLabel)-1;

if (OpCodeAttr::HasFallThrough(op))

{

// emit jump around.

jumpAroundLabel = this->DefineLabel();

this->Br(jumpAroundLabel);

}

// emit the long jump

this->MarkLabel(longJumpLabel);

this->BrLong(Js::OpCode::BrLong, labelId);

if (jumpAroundLabel != (ByteCodeLabel)-1)

{

this->MarkLabel(jumpAroundLabel);

}

return;

}

}

}

#endif

//

// Branch targets are created in two passes:

// - In the instruction stream, write "labelID" into "OpLayoutBrC.Offset". Record this

// location in "m\_jumpOffsets" to be patched later.

// - When the byte-code is closed, update all "OpLayoutBrC.Offset"'s with their actual

// destinations.

//

JumpInfo jumpInfo = { labelId, jumpByteOffset };

m\_jumpOffsets->Add(jumpInfo);

}

#ifdef BYTECODE\_BRANCH\_ISLAND

int32 ByteCodeWriter::GetBranchLimit()

{

#ifdef BYTECODE\_TESTING

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

{

// minimum 64

return min(max(Js::Configuration::Global.flags.ByteCodeBranchLimit, 64), SHRT\_MAX + 1);

}

#endif

return SHRT\_MAX + 1;

}

void ByteCodeWriter::AddLongJumpOffset(ByteCodeLabel labelId, uint fieldByteOffsetFromEnd) // Offset of "Offset" field in OpLayout, in bytes

{

Assert(useBranchIsland);

AssertMsg(fieldByteOffsetFromEnd < 100, "Ensure valid field offset");

//

// Branch targets are created in two passes:

// - In the instruction stream, write "labelID" into "OpLayoutBrC.Offset". Record this

// location in "m\_jumpOffsets" to be patched later.

// - When the byte-code is closed, update all "OpLayoutBrC.Offset"'s with their actual

// destinations.

//

uint jumpByteOffset = m\_byteCodeData.GetCurrentOffset() - fieldByteOffsetFromEnd;

JumpInfo jumpInfo = { labelId, jumpByteOffset };

m\_longJumpOffsets->Add(jumpInfo);

}

void ByteCodeWriter::BrLong(OpCode op, ByteCodeLabel labelID)

{

Assert(useBranchIsland);

CheckOpen();

CheckOp(op, OpLayoutType::BrLong);

CheckLabel(labelID);

Assert(!OpCodeAttr::HasMultiSizeLayout(op));

size\_t const offsetOfRelativeJumpOffsetFromEnd = sizeof(OpLayoutBrLong) - offsetof(OpLayoutBrLong, RelativeJumpOffset);

OpLayoutBrLong data;

data.RelativeJumpOffset = offsetOfRelativeJumpOffsetFromEnd;

m\_byteCodeData.Encode(op, &data, sizeof(data), this);

AddLongJumpOffset(labelID, offsetOfRelativeJumpOffsetFromEnd);

}

void ByteCodeWriter::UpdateNextBranchIslandOffset(uint firstUnknownJumpInfo, uint firstUnknownJumpOffset)

{

this->firstUnknownJumpInfo = firstUnknownJumpInfo;

// We will need to emit the next branch from the first branch + branch limit.

// But leave room for the jump around and one extra byte code instruction.

// Also account for all the long branches we may have to emit as well.

this->nextBranchIslandOffset = firstUnknownJumpOffset + GetBranchLimit()

- JumpAroundSize - MaxLayoutSize - MaxOpCodeSize - LongBranchSize \* (m\_jumpOffsets->Count() - firstUnknownJumpInfo);

}

void ByteCodeWriter::EnsureLongBranch(Js::OpCode op)

{

Assert(useBranchIsland);

int currentOffset = this->m\_byteCodeData.GetCurrentOffset();

// See if we need to emit branch island yet, and avoid recursion.

if (currentOffset < this->nextBranchIslandOffset || this->inEnsureLongBranch)

{

lastOpcode = op;

return;

}

// Leave actually may continue right after, it is only no fall through in the JIT.

bool needBranchAround = OpCodeAttr::HasFallThrough(lastOpcode) || lastOpcode == Js::OpCode::Leave;

lastOpcode = op;

// If we are about to emit a no fall through op and the last was has fall through

// then just emit the no fall through op, and then we can skip the branch around.

// Except at label or StatementBoundary, we always want to emit before them.

if ((needBranchAround && !OpCodeAttr::HasFallThrough(op))

&& op != Js::OpCode::StatementBoundary && op != Js::OpCode::Label)

{

return;

}

ByteCodeLabel branchAroundLabel = (Js::ByteCodeLabel)-1;

bool foundUnknown = m\_jumpOffsets->MapUntilFrom(firstUnknownJumpInfo,

[=, &branchAroundLabel, &currentOffset](int index, JumpInfo& jumpInfo)

{

//

// Read "labelID" stored at the offset within the byte-code.

//

uint jumpByteOffset = jumpInfo.patchOffset;

AssertMsg(jumpByteOffset <= this->m\_byteCodeData.GetCurrentOffset() - sizeof(JumpOffset),

"Must have valid jump site within byte-code to back-patch");

ByteCodeLabel labelID = jumpInfo.labelId;

CheckLabel(labelID);

// See if the label has bee marked yet.

uint const labelByteOffset = m\_labelOffsets->Item(labelID);

if (labelByteOffset != UINT\_MAX)

{

// If a label is already defined, then it should be short

// (otherwise we should have emitted a branch island for it already).

Assert((int)labelByteOffset - (int)jumpByteOffset < GetBranchLimit()

&& (int)labelByteOffset - (int)jumpByteOffset >= -GetBranchLimit());

return false;

}

this->UpdateNextBranchIslandOffset(index, jumpByteOffset);

// Flush all the jump that are half of the way to the limit as well so we don't have

// as many jump around of branch island.

int flushNextBranchIslandOffset = this->nextBranchIslandOffset - GetBranchLimit() / 2;

if (currentOffset < flushNextBranchIslandOffset)

{

// No need to for long branch yet. Terminate the loop.

return true;

}

if (labelID == branchAroundLabel)

{

// Let's not flush the branchAroundLabel.

// Should happen very rarely and mostly when the branch limit is very small.

// This should be the last short jump we have just emitted (below).

Assert(index == m\_jumpOffsets->Count() - 1);

Assert(currentOffset < this->nextBranchIslandOffset);

return true;

}

// Emit long branch

// Prevent recursion when we emit byte code here

this->inEnsureLongBranch = true;

// Create the branch label and update the jumpInfo.

// Need to update the jumpInfo before we add the branch island as that might resize the m\_jumpOffsets list.

ByteCodeLabel longBranchLabel = this->DefineLabel();

jumpInfo.labelId = longBranchLabel;

// Emit the branch around if it hasn't been emitted already

if (branchAroundLabel == (Js::ByteCodeLabel)-1 && needBranchAround)

{

branchAroundLabel = this->DefineLabel();

this->Br(Js::OpCode::Br, branchAroundLabel);

Assert(this->m\_byteCodeData.GetCurrentOffset() - currentOffset == JumpAroundSize);

currentOffset += JumpAroundSize;

// Continue to count he jumpAroundSize, because we may have to emit

// yet another branch island right after if the jumpAroundSize is included.

}

// Emit the long branch

this->MarkLabel(longBranchLabel);

this->BrLong(Js::OpCode::BrLong, labelID);

this->inEnsureLongBranch = false;

Assert(this->m\_byteCodeData.GetCurrentOffset() - currentOffset == LongBranchSize);

currentOffset += LongBranchSize;

return false;

});

if (!foundUnknown)

{

// Nothing is found, just set the next branch island from the current offset

this->UpdateNextBranchIslandOffset(this->m\_jumpOffsets->Count(), currentOffset);

}

if (branchAroundLabel != (Js::ByteCodeLabel)-1)

{

// Make the branch around label if we needed one

this->MarkLabel(branchAroundLabel);

}

}

#endif

void ByteCodeWriter::StartStatement(ParseNode\* node, uint32 tmpRegCount)

{

if (m\_pMatchingNode)

{

if (m\_pMatchingNode == node)

{

m\_matchingNodeRefCount++;

}

return;

}

#ifdef BYTECODE\_BRANCH\_ISLAND

if (useBranchIsland)

{

// If we are going to emit a branch island, it should be before the statement start

this->EnsureLongBranch(Js::OpCode::StatementBoundary);

}

#endif

m\_pMatchingNode = node;

m\_beginCodeSpan = m\_byteCodeData.GetCurrentOffset();

if (m\_isInDebugMode && m\_tmpRegCount != tmpRegCount)

{

Unsigned1(OpCode::EmitTmpRegCount, tmpRegCount);

m\_tmpRegCount = tmpRegCount;

}

}

void ByteCodeWriter::EndStatement(ParseNode\* node)

{

AssertMsg(m\_pMatchingNode, "EndStatement unmatched to StartStartment");

if (m\_pMatchingNode != node)

{

return;

}

else if (m\_matchingNodeRefCount > 0)

{

m\_matchingNodeRefCount--;

return;

}

if (m\_byteCodeData.GetCurrentOffset() != m\_beginCodeSpan)

{

if (m\_isInDebugMode)

{

FunctionBody::StatementMap\* pCurrentStatement = FunctionBody::StatementMap::New(this->m\_functionWrite->GetScriptContext()->GetRecycler());

if (pCurrentStatement)

{

pCurrentStatement->sourceSpan.begin = node->ichMin;

pCurrentStatement->sourceSpan.end = node->ichLim;

pCurrentStatement->byteCodeSpan.begin = m\_beginCodeSpan;

pCurrentStatement->byteCodeSpan.end = m\_byteCodeData.GetCurrentOffset() - 1;

m\_functionWrite->RecordStatementMap(pCurrentStatement);

}

}

else

{

StatementData currentStatement;

currentStatement.sourceBegin = node->ichMin;

currentStatement.bytecodeBegin = m\_beginCodeSpan;

m\_functionWrite->RecordStatementMap(spanIter, &currentStatement);

}

}

m\_pMatchingNode = nullptr;

}

void ByteCodeWriter::StartSubexpression(ParseNode\* node)

{

if (!m\_isInDebugMode || !m\_pMatchingNode) // Subexpression not in debug mode or not enclosed in regular statement

{

return;

}

#ifdef BYTECODE\_BRANCH\_ISLAND

// If we are going to emit a branch island, it should be before the statement start

this->EnsureLongBranch(Js::OpCode::StatementBoundary);

#endif

m\_subexpressionNodesStack->Push(SubexpressionNode(node, m\_byteCodeData.GetCurrentOffset()));

}

void ByteCodeWriter::EndSubexpression(ParseNode\* node)

{

if (!m\_isInDebugMode || m\_subexpressionNodesStack->Empty() || m\_subexpressionNodesStack->Peek().node != node)

{

return;

}

if (m\_byteCodeData.GetCurrentOffset() != m\_beginCodeSpan)

{

FunctionBody::StatementMap\* pCurrentStatement = FunctionBody::StatementMap::New(this->m\_functionWrite->GetScriptContext()->GetRecycler());

if (pCurrentStatement)

{

pCurrentStatement->sourceSpan.begin = node->ichMin;

pCurrentStatement->sourceSpan.end = node->ichLim;

SubexpressionNode subexpressionNode = m\_subexpressionNodesStack->Pop();

pCurrentStatement->byteCodeSpan.begin = subexpressionNode.beginCodeSpan;

pCurrentStatement->byteCodeSpan.end = m\_byteCodeData.GetCurrentOffset() - 1;

pCurrentStatement->isSubexpression = true;

m\_functionWrite->RecordStatementMap(pCurrentStatement);

}

}

}

// Pushes a new debugger scope onto the stack. This information is used when determining

// what the current scope is for tracking of let/const initialization offsets (for detecting

// dead zones).

void ByteCodeWriter::PushDebuggerScope(Js::DebuggerScope\* debuggerScope)

{

Assert(debuggerScope);

debuggerScope->SetParentScope(m\_currentDebuggerScope);

m\_currentDebuggerScope = debuggerScope;

OUTPUT\_VERBOSE\_TRACE(Js::DebuggerPhase, L"PushDebuggerScope() - Pushed scope 0x%p of type %d.\n", m\_currentDebuggerScope, m\_currentDebuggerScope->scopeType);

}

// Pops the current debugger scope from the stack.

void ByteCodeWriter::PopDebuggerScope()

{

Assert(m\_currentDebuggerScope);

OUTPUT\_VERBOSE\_TRACE(Js::DebuggerPhase, L"PopDebuggerScope() - Popped scope 0x%p of type %d.\n", m\_currentDebuggerScope, m\_currentDebuggerScope->scopeType);

if (m\_currentDebuggerScope != nullptr)

{

m\_currentDebuggerScope = m\_currentDebuggerScope->GetParentScope();

}

}

DebuggerScope\* ByteCodeWriter::RecordStartScopeObject(DiagExtraScopesType scopeType, RegSlot scopeLocation, int\* index)

{

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

{

scopeLocation = ConsumeReg(scopeLocation);

}

DebuggerScope\* debuggerScope = m\_functionWrite->RecordStartScopeObject(scopeType, m\_byteCodeData.GetCurrentOffset(), scopeLocation, index);

PushDebuggerScope(debuggerScope);

return debuggerScope;

}

void ByteCodeWriter::AddPropertyToDebuggerScope(

DebuggerScope\* debuggerScope,

RegSlot location,

Js::PropertyId propertyId,

bool shouldConsumeRegister /\*= true\*/,

DebuggerScopePropertyFlags flags /\*= DebuggerScopePropertyFlags\_None\*/,

bool isFunctionDeclaration /\*= false\*/)

{

Assert(debuggerScope);

// Activation object doesn't use register and slot array location represents the

// index in the array. Only need to consume for register slots.

if (shouldConsumeRegister)

{

Assert(location != Js::Constants::NoRegister);

location = ConsumeReg(location);

}

debuggerScope->AddProperty(location, propertyId, flags);

// Only need to update properties in debug mode (even for slot array, which is tracked in non-debug mode,

// since the offset is only used for debugging).

if (this->m\_isInDebugMode && isFunctionDeclaration)

{

AssertMsg(this->m\_currentDebuggerScope, "Function declarations can only be added in a block scope.");

AssertMsg(debuggerScope == this->m\_currentDebuggerScope

|| debuggerScope == this->m\_currentDebuggerScope->siblingScope,

"Function declarations should always be added to the current scope.");

// If this is a function declaration, it doesn't have a dead zone region so

// we just update its byte code initialization offset to the start of the block.

this->UpdateDebuggerPropertyInitializationOffset(

debuggerScope,

location,

propertyId,

false /\*shouldConsumeRegister\*/, // Register would have already been consumed above, if needed.

debuggerScope->GetStart(),

isFunctionDeclaration);

}

}

void ByteCodeWriter::RecordEndScopeObject()

{

Assert(this->m\_currentDebuggerScope);

m\_functionWrite->RecordEndScopeObject(this->m\_currentDebuggerScope, m\_byteCodeData.GetCurrentOffset() - 1);

PopDebuggerScope();

}

void ByteCodeWriter::UpdateDebuggerPropertyInitializationOffset(

Js::DebuggerScope\* currentDebuggerScope,

Js::RegSlot location,

Js::PropertyId propertyId,

bool shouldConsumeRegister/\* = true\*/,

int byteCodeOffset/\* = Constants::InvalidOffset\*/,

bool isFunctionDeclaration /\*= false\*/)

{

#if DBG

bool isInDebugMode = m\_isInDebugMode

#if DBG\_DUMP

|| Js::Configuration::Global.flags.Debug

#endif // DBG\_DUMP

;

AssertMsg(isInDebugMode, "Property offsets should only ever be updated in debug mode (not used in non-debug).");

#endif // DBG

Assert(currentDebuggerScope);

if (shouldConsumeRegister)

{

Assert(location != Js::Constants::NoRegister);

location = ConsumeReg(location);

}

if (byteCodeOffset == Constants::InvalidOffset)

{

// Use the current offset if no offset is passed in.

byteCodeOffset = this->m\_byteCodeData.GetCurrentOffset();

}

// Search through the scope chain starting with the current up through the parents to see if the

// property can be found and updated.

while (currentDebuggerScope != nullptr)

{

if (currentDebuggerScope->UpdatePropertyInitializationOffset(location, propertyId, byteCodeOffset, isFunctionDeclaration))

{

break;

}

currentDebuggerScope = currentDebuggerScope->GetParentScope();

}

}

void ByteCodeWriter::RecordFrameDisplayRegister(RegSlot slot)

{

slot = ConsumeReg(slot);

m\_functionWrite->RecordFrameDisplayRegister(slot);

}

void ByteCodeWriter::RecordObjectRegister(RegSlot slot)

{

slot = ConsumeReg(slot);

m\_functionWrite->RecordObjectRegister(slot);

}

void ByteCodeWriter::RecordStatementAdjustment(FunctionBody::StatementAdjustmentType type)

{

if (m\_isInDebugMode)

{

m\_functionWrite->RecordStatementAdjustment(m\_byteCodeData.GetCurrentOffset(), type);

}

}

void ByteCodeWriter::RecordCrossFrameEntryExitRecord(bool isEnterBlock)

{

if (m\_isInDebugMode)

{

m\_functionWrite->RecordCrossFrameEntryExitRecord(m\_byteCodeData.GetCurrentOffset(), isEnterBlock);

}

}

void ByteCodeWriter::RecordForInOrOfCollectionScope()

{

if (m\_isInDebugMode && this->m\_currentDebuggerScope != nullptr)

{

this->m\_currentDebuggerScope->UpdatePropertiesInForInOrOfCollectionScope();

}

}

uint ByteCodeWriter::EnterLoop(Js::ByteCodeLabel loopEntrance)

{

#ifdef BYTECODE\_BRANCH\_ISLAND

if (useBranchIsland)

{

// If we are going to emit a branch island, it should be before the loop header

this->EnsureLongBranch(Js::OpCode::StatementBoundary);

}

#endif

uint loopId = m\_functionWrite->IncrLoopCount();

Assert((uint)m\_loopHeaders->Count() == loopId);

m\_loopHeaders->Add(LoopHeaderData(m\_byteCodeData.GetCurrentOffset(), 0, m\_loopNest > 0));

m\_loopNest++;

m\_functionWrite->SetHasNestedLoop(m\_loopNest > 1);

Js::OpCode loopBodyOpcode = Js::OpCode::LoopBodyStart;

#if ENABLE\_PROFILE\_INFO

if (Js::DynamicProfileInfo::EnableImplicitCallFlags(GetFunctionWrite()))

{

this->Unsigned1(Js::OpCode::ProfiledLoopStart, loopId);

loopBodyOpcode = Js::OpCode::ProfiledLoopBodyStart;

}

#endif

this->MarkLabel(loopEntrance);

if (this->DoJitLoopBodies() || this->DoInterruptProbes())

{

this->Unsigned1(loopBodyOpcode, loopId);

}

return loopId;

}

void ByteCodeWriter::ExitLoop(uint loopId)

{

#if ENABLE\_PROFILE\_INFO

if (Js::DynamicProfileInfo::EnableImplicitCallFlags(GetFunctionWrite()))

{

this->Unsigned1(Js::OpCode::ProfiledLoopEnd, loopId);

}

#endif

Assert(m\_loopNest > 0);

m\_loopNest--;

m\_loopHeaders->Item(loopId).endOffset = m\_byteCodeData.GetCurrentOffset();

}

void ByteCodeWriter::IncreaseByteCodeCount()

{

m\_byteCodeCount++;

if (m\_loopNest > 0)

{

m\_byteCodeInLoopCount++;

}

}

void ByteCodeWriter::Data::Create(uint initSize, ArenaAllocator\* tmpAlloc)

{

//

// Allocate the initial byte-code block to write into.

//

tempAllocator = tmpAlloc;

AssertMsg(head == nullptr, "Missing dispose?");

currentOffset = 0;

head = Anew(tempAllocator, DataChunk, tempAllocator, initSize);

current = head;

}

void ByteCodeWriter::Data::Reset()

{

currentOffset = 0;

DataChunk\* currentChunk = head;

while (currentChunk)

{

// reset to the starting point

currentChunk->Reset();

currentChunk = currentChunk->nextChunk;

}

current = head;

}

void ByteCodeWriter::Data::SetCurrent(uint offset, DataChunk\* currChunk)

{

this->current = currChunk;

this->currentOffset = offset;

}

/// Copies its contents to a final contiguous section of memory.

void ByteCodeWriter::Data::Copy(Recycler\* alloc, ByteBlock \*\* finalBlock)

{

AssertMsg(finalBlock != nullptr, "Must have valid storage");

uint cbFinalData = GetCurrentOffset();

if (cbFinalData == 0)

{

\*finalBlock = nullptr;

}

else

{

ByteBlock\* finalByteCodeBlock = ByteBlock::New(alloc, /\*intialContent\*/nullptr, cbFinalData);

DataChunk\* currentChunk = head;

size\_t bytesLeftToCopy = cbFinalData;

byte\* currentDest = finalByteCodeBlock->GetBuffer();

while (true)

{

if (bytesLeftToCopy <= currentChunk->GetSize())

{

js\_memcpy\_s(currentDest, bytesLeftToCopy, currentChunk->GetBuffer(), bytesLeftToCopy);

break;

}

js\_memcpy\_s(currentDest, bytesLeftToCopy, currentChunk->GetBuffer(), currentChunk->GetSize());

bytesLeftToCopy -= currentChunk->GetSize();

currentDest += currentChunk->GetSize();

currentChunk = currentChunk->nextChunk;

AssertMsg(currentChunk, "We are copying more data than we have!");

}

\*finalBlock = finalByteCodeBlock;

}

}

template <>

\_\_inline uint ByteCodeWriter::Data::EncodeT<SmallLayout>(OpCode op, ByteCodeWriter\* writer)

{

#ifdef BYTECODE\_BRANCH\_ISLAND

if (writer->useBranchIsland)

{

writer->EnsureLongBranch(op);

}

#endif

Assert(op < Js::OpCode::ByteCodeLast);

Assert(!OpCodeAttr::BackEndOnly(op));

uint offset;

if (op <= Js::OpCode::MaxByteSizedOpcodes)

{

byte byteop = (byte)op;

offset = Write(&byteop, sizeof(byte));

}

else

{

byte byteop = (byte)Js::OpCode::ExtendedOpcodePrefix;

offset = Write(&byteop, sizeof(byte));

byteop = (byte)op;

Write(&byteop, sizeof(byte));

}

if (op != Js::OpCode::Ld\_A)

{

writer->m\_byteCodeWithoutLDACount++;

}

writer->IncreaseByteCodeCount();

return offset;

}

template <LayoutSize layoutSize>

\_\_inline uint ByteCodeWriter::Data::EncodeT(OpCode op, ByteCodeWriter\* writer)

{

#ifdef BYTECODE\_BRANCH\_ISLAND

if (writer->useBranchIsland)

{

writer->EnsureLongBranch(op);

}

#endif

Assert(op < Js::OpCode::ByteCodeLast);

Assert(!OpCodeAttr::BackEndOnly(op));

Assert(OpCodeAttr::HasMultiSizeLayout(op));

CompileAssert(layoutSize != SmallLayout);

const byte exop = (byte)((op <= Js::OpCode::MaxByteSizedOpcodes) ?

(layoutSize == LargeLayout ? Js::OpCode::LargeLayoutPrefix : Js::OpCode::MediumLayoutPrefix) :

(layoutSize == LargeLayout ? Js::OpCode::ExtendedLargeLayoutPrefix : Js::OpCode::ExtendedMediumLayoutPrefix));

uint offset = Write(&exop, sizeof(byte));

Write(&op, sizeof(byte));

if (op != Js::OpCode::Ld\_A)

{

writer->m\_byteCodeWithoutLDACount++;

}

writer->IncreaseByteCodeCount();

return offset;

}

template <LayoutSize layoutSize>

\_\_inline uint ByteCodeWriter::Data::EncodeT(OpCode op, const void\* rawData, int byteSize, ByteCodeWriter\* writer)

{

AssertMsg((rawData != nullptr) && (byteSize < 100), "Ensure valid data for opcode");

uint offset = EncodeT<layoutSize>(op, writer);

Write(rawData, byteSize);

return offset;

}

\_\_inline uint ByteCodeWriter::Data::Encode(const void\* rawData, int byteSize)

{

AssertMsg(rawData != nullptr, "Ensure valid data for opcode");

return Write(rawData, byteSize);

}

\_\_inline uint ByteCodeWriter::Data::Write(\_\_in\_bcount(byteSize) const void\* data, \_\_in uint byteSize)

{

// Simple case where the current chunk has enough space.

uint bytesFree = current->RemainingBytes();

if (bytesFree >= byteSize)

{

current->WriteUnsafe(data, byteSize);

}

else

{

SlowWrite(data, byteSize);

}

uint offset = currentOffset;

currentOffset = offset + byteSize;

return offset;

}

/// Requires buffer extension.

\_\_declspec(noinline) void ByteCodeWriter::Data::SlowWrite(\_\_in\_bcount(byteSize) const void\* data, \_\_in uint byteSize)

{

AssertMsg(byteSize > current->RemainingBytes(), "We should not need an extension if there is enough space in the current chunk");

uint bytesLeftToWrite = byteSize;

byte\* dataToBeWritten = (byte\*)data;

// the next chunk may already be created in the case that we are patching bytecode.

// If so, we want to move the pointer to the beginning of the buffer

if (current->nextChunk)

{

current->nextChunk->SetCurrentOffset(0);

}

while (true)

{

uint bytesFree = current->RemainingBytes();

if (bytesFree >= bytesLeftToWrite)

{

current->WriteUnsafe(dataToBeWritten, bytesLeftToWrite);

break;

}

current->WriteUnsafe(dataToBeWritten, bytesFree);

bytesLeftToWrite -= bytesFree;

dataToBeWritten += bytesFree;

// Create a new chunk when needed

if (!current->nextChunk)

{

AddChunk(bytesLeftToWrite);

}

current = current->nextChunk;

}

}

void ByteCodeWriter::Data::AddChunk(uint byteSize)

{

AssertMsg(current->nextChunk == nullptr, "Do we really need to grow?");

// For some data elements i.e. bytecode we have a good initial size and

// therefore, we use a conservative growth strategy - and grow by a fixed size.

uint newSize = fixedGrowthPolicy ? max(byteSize, static\_cast<uint>(3 \* AutoSystemInfo::PageSize)) : max(byteSize, static\_cast<uint>(current->GetSize() \* 2));

DataChunk\* newChunk = Anew(tempAllocator, DataChunk, tempAllocator, newSize);

current->nextChunk = newChunk;

}

#if DBG\_DUMP

uint ByteCodeWriter::ByteCodeDataSize()

{

return m\_byteCodeData.GetCurrentOffset();

}

uint ByteCodeWriter::AuxiliaryDataSize()

{

return m\_auxiliaryData.GetCurrentOffset();

}

uint ByteCodeWriter::AuxiliaryContextDataSize()

{

return m\_auxContextData.GetCurrentOffset();

}

#endif

} // 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.

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

#pragma once

namespace Js

{

struct ByteCodeWriter

{

protected:

struct DataChunk

{

private:

byte\* currentByte;

byte\* buffer;

uint byteSize;

public:

DataChunk\* nextChunk;

DataChunk(ArenaAllocator\* allocator, uint initSize) :

nextChunk(nullptr),

byteSize(initSize)

{

buffer = AnewArray(allocator, byte, initSize);

currentByte = buffer;

}

inline uint GetSize()

{

return byteSize;

}

inline const byte\* GetBuffer()

{

return buffer;

}

inline void Reset()

{

currentByte = buffer;

}

inline uint RemainingBytes()

{

Assert(byteSize >= GetCurrentOffset());

return byteSize - GetCurrentOffset();

}

inline uint GetCurrentOffset()

{

Assert(currentByte >= buffer);

return (uint) (currentByte - buffer);

}

inline void SetCurrentOffset(uint offset)

{

currentByte = buffer + offset;

}

// This does not do check if there is enough space for the copy to succeed.

\_\_inline void WriteUnsafe(\_\_in\_bcount(byteSize) const void\* data, \_\_in uint byteSize)

{

AssertMsg(RemainingBytes() >= byteSize, "We do not have enough room");

js\_memcpy\_s(currentByte, this->RemainingBytes(), data, byteSize);

currentByte += byteSize;

}

};

// This is a linked list of data chunks. It is designed to be allocated

// once and re-used. After a call to Reset(), this data structure can be reused to

// store more data.

struct Data

{

private:

ArenaAllocator\* tempAllocator;

DataChunk\* head; // First chunk to be written to

DataChunk\* current; // The current chunk being written to

uint currentOffset; // The global offset of last byte written to in the linked data structure

bool fixedGrowthPolicy;

\_\_inline uint Write(\_\_in\_bcount(byteSize) const void\* data, \_\_in uint byteSize);

\_\_declspec(noinline) void SlowWrite(\_\_in\_bcount(byteSize) const void\* data, \_\_in uint byteSize);

void AddChunk(uint byteSize);

public:

Data(bool fixedGrowthPolicy = false) : head(nullptr),

current(nullptr),

tempAllocator(nullptr),

currentOffset(0),

fixedGrowthPolicy(fixedGrowthPolicy)

{

}

void Create(uint initSize, ArenaAllocator\* tmpAlloc);

inline uint GetCurrentOffset() const { return currentOffset; }

inline DataChunk \* GetCurrentChunk() const { return &(\*current); }

void SetCurrent(uint offset, DataChunk\* currChunk);

void Copy(Recycler\* alloc, ByteBlock \*\* finalBlock);

uint Encode(OpCode op, ByteCodeWriter\* writer)

{

return EncodeT<Js::SmallLayout>(op, writer);

}

uint Encode(OpCode op, const void \* rawData, int byteSize, ByteCodeWriter\* writer)

{

return EncodeT<Js::SmallLayout>(op, rawData, byteSize, writer);

}

uint Encode(const void \* rawData, int byteSize);

template <LayoutSize layoutSize> uint EncodeT(OpCode op, ByteCodeWriter\* writer);

template <> uint EncodeT<SmallLayout>(OpCode op, ByteCodeWriter\* writer);

template <LayoutSize layoutSize> uint EncodeT(OpCode op, const void \* rawData, int byteSize, ByteCodeWriter\* writer);

// asm.js encoding

uint Encode(OpCodeAsmJs op, ByteCodeWriter\* writer){ return EncodeT<Js::SmallLayout>(op, writer); }

uint Encode(OpCodeAsmJs op, const void \* rawData, int byteSize, ByteCodeWriter\* writer, bool isPatching = false){ return EncodeT<Js::SmallLayout>(op, rawData, byteSize, writer, isPatching); }

template <LayoutSize layoutSize> uint EncodeT(OpCodeAsmJs op, ByteCodeWriter\* writer, bool isPatching = false);

template <> uint EncodeT<SmallLayout>(OpCodeAsmJs op, ByteCodeWriter\* writer, bool isPatching);

template <LayoutSize layoutSize> uint EncodeT(OpCodeAsmJs op, const void \* rawData, int byteSize, ByteCodeWriter\* writer, bool isPatching = false);

void Reset();

};

struct LoopHeaderData {

uint startOffset;

uint endOffset;

bool isNested;

LoopHeaderData() {}

LoopHeaderData(uint startOffset, uint endOffset, bool isNested) : startOffset(startOffset), endOffset(endOffset), isNested(isNested){}

};

JsUtil::List<uint, ArenaAllocator> \* m\_labelOffsets; // Label offsets, once defined

struct JumpInfo

{

ByteCodeLabel labelId;

uint patchOffset;

};

#ifdef BYTECODE\_BRANCH\_ISLAND

bool useBranchIsland;

JsUtil::List<JumpInfo, ArenaAllocator> \* m\_longJumpOffsets;

Js::OpCode lastOpcode;

bool inEnsureLongBranch;

uint firstUnknownJumpInfo;

int nextBranchIslandOffset;

// Size of emitting a jump around byte code instruction

static size\_t const JumpAroundSize = sizeof(byte) + sizeof(OpLayoutBr);

// Size of emitting a long jump byte code instruction

CompileAssert(OpCodeInfo<Js::OpCode::BrLong>::IsExtendedOpcode); // extended opcode, opcode size is always sizeof(OpCode)

static size\_t const LongBranchSize = sizeof(OpCode) + sizeof(OpLayoutBrLong);

#endif

JsUtil::List<JumpInfo, ArenaAllocator> \* m\_jumpOffsets; // Offsets to replace "ByteCodeLabel" with actual destination

JsUtil::List<LoopHeaderData, ArenaAllocator> \* m\_loopHeaders; // Start/End offsets for loops

SListBase<size\_t> rootObjectLoadInlineCacheOffsets; // load inline cache offsets

SListBase<size\_t> rootObjectStoreInlineCacheOffsets; // load inline cache offsets

SListBase<size\_t> rootObjectLoadMethodInlineCacheOffsets;

FunctionBody\* m\_functionWrite; // Function being written

Data m\_byteCodeData; // Accumulated byte-code

Data m\_auxiliaryData; // Optional accumulated auxiliary data

Data m\_auxContextData; // Optional accumulated auxiliary ScriptContext specific data

uint m\_beginCodeSpan; // Debug Info for where in bytecode the current statement starts. (Statements do not nest.)

ParseNode\* m\_pMatchingNode; // Parse node for statement we are tracking debugInfo for.

// This ref count mechanism ensures that nested Start/End Statement with matching parse node will not mess-up the upper Start/End balance.

// This count will be incremented/decremented when the nested Start/End has the same m\_pMatchingNode.

int m\_matchingNodeRefCount;

struct SubexpressionNode

{

ParseNode\* node;

int beginCodeSpan;

SubexpressionNode() {}

SubexpressionNode(ParseNode\* node, int beginCodeSpan) : node(node), beginCodeSpan(beginCodeSpan) {}

};

JsUtil::Stack<SubexpressionNode> \* m\_subexpressionNodesStack; // Parse nodes for Subexpressions not participating in debug-stepping actions

SmallSpanSequenceIter spanIter;

int m\_loopNest;

uint m\_byteCodeCount;

uint m\_byteCodeWithoutLDACount; // Number of total bytecodes except LD\_A and LdUndef

uint m\_byteCodeInLoopCount;

uint32 m\_tmpRegCount;

bool m\_doJitLoopBodies;

bool m\_hasLoop;

bool m\_isInDebugMode;

bool m\_doInterruptProbe;

public:

struct CacheIdUnit {

uint cacheId;

bool isRootObjectCache;

CacheIdUnit() {}

CacheIdUnit(uint cacheId, bool isRootObjectCache = false) : cacheId(cacheId), isRootObjectCache(isRootObjectCache) {}

};

protected:

// A map of, bytecode register in which the function to be called is, to the inline cache index associated with the load of the function.

typedef JsUtil::BaseDictionary<Js::RegSlot, CacheIdUnit, ArenaAllocator, PrimeSizePolicy> CallRegToLdFldCacheIndexMap;

CallRegToLdFldCacheIndexMap\* callRegToLdFldCacheIndexMap;

// Debugger fields.

Js::DebuggerScope\* m\_currentDebuggerScope;

#if DBG

bool isInitialized;

bool isInUse;

#endif

void IncreaseByteCodeCount();

void AddJumpOffset(Js::OpCode op, ByteCodeLabel labelId, uint fieldByteOffset);

RegSlot ConsumeReg(RegSlot reg);

inline void CheckOpen();

inline void CheckLabel(ByteCodeLabel labelID);

inline void CheckOp(OpCode op, OpLayoutType layoutType);

inline void CheckReg(RegSlot registerID);

bool DoDynamicProfileOpcode(Phase tag, bool noHeuristics = false) const;

template <typename T>

void PatchJumpOffset(JsUtil::List<JumpInfo, ArenaAllocator> \* jumpOffsets, byte \* byteBuffer, uint byteCount);

#ifdef BYTECODE\_BRANCH\_ISLAND

static int32 GetBranchLimit();

void AddLongJumpOffset(ByteCodeLabel labelId, uint fieldByteOffset);

void EnsureLongBranch(Js::OpCode op);

void UpdateNextBranchIslandOffset(uint firstUnknownJumpInfo, uint firstUnknownJumpOffset);

#endif

// Debugger methods.

void PushDebuggerScope(Js::DebuggerScope\* debuggerScope);

void PopDebuggerScope();

public:

ByteCodeWriter() : m\_byteCodeCount(/\*fixedGrowthPolicy=\*/ true), m\_currentDebuggerScope(nullptr) {}

#if DBG

~ByteCodeWriter() { Assert(!isInUse); }

#endif

void Create();

void InitData(ArenaAllocator\* alloc, long initCodeBufferSize);

void Begin(ByteCodeGenerator\* byteCodeGenerator, FunctionBody\* functionWrite, ArenaAllocator\* alloc, bool doJitLoopBodies, bool hasLoop);

#ifdef LOG\_BYTECODE\_AST\_RATIO

void End(long currentAstSize, long maxAstSize);

#else

void End();

#endif

void Reset();

void AllocateLoopHeaders();

#if DBG\_DUMP

uint ByteCodeDataSize();

uint AuxiliaryDataSize();

uint AuxiliaryContextDataSize();

#endif

void Empty(OpCode op);

void Reg1(OpCode op, RegSlot R0);

void Reg2(OpCode op, RegSlot R0, RegSlot R1);

void Reg3(OpCode op, RegSlot R0, RegSlot R1, RegSlot R2);

void Reg3C(OpCode op, RegSlot R0, RegSlot R1, RegSlot R2, uint cacheId);

void Reg4(OpCode op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3);

void Reg1Unsigned1(OpCode op, RegSlot R0, uint C1);

void Reg2B1(OpCode op, RegSlot R0, RegSlot R1, uint8 B3);

void Reg3B1(OpCode op, RegSlot R0, RegSlot R1, RegSlot R2, uint8 B3);

void Reg5(OpCode op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3, RegSlot R4);

void ArgIn0(RegSlot arg);

template <bool isVar>

void ArgOut(ArgSlot arg, RegSlot reg, ProfileId callSiteId);

void ArgOutEnv(ArgSlot arg);

#ifdef BYTECODE\_BRANCH\_ISLAND

void BrLong(OpCode op, ByteCodeLabel labelID);

#endif

void Br(ByteCodeLabel labelID);

void Br(OpCode op, ByteCodeLabel labelID);

void BrReg1(OpCode op, ByteCodeLabel labelID, RegSlot R1);

void BrS(OpCode op, ByteCodeLabel labelID, byte val);

void BrReg2(OpCode op, ByteCodeLabel labelID, RegSlot R1, RegSlot R2);

void BrProperty(OpCode op, ByteCodeLabel labelID, RegSlot R1, PropertyIdIndexType propertyIdIndex);

void BrLocalProperty(OpCode op, ByteCodeLabel labelID, PropertyIdIndexType propertyIdIndex);

void BrEnvProperty(OpCode op, ByteCodeLabel labelID, PropertyIdIndexType propertyIdIndex, int32 slotIndex);

void StartCall(OpCode op, ArgSlot ArgCount);

void CallI(OpCode op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, ProfileId callSiteId, CallFlags callFlags = CallFlags\_None);

void CallIExtended(OpCode op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, CallIExtendedOptions options, const void \*buffer, uint byteCount, ProfileId callSiteId, CallFlags callFlags = CallFlags\_None);

void RemoveEntryForRegSlotFromCacheIdMap(RegSlot functionRegister);

void Element(OpCode op, RegSlot value, RegSlot instance, RegSlot element, bool instanceAtReturnRegOK = false);

void ElementUnsigned1(OpCode op, RegSlot value, RegSlot instance, uint32 element);

void Property(OpCode op, RegSlot Value, RegSlot Instance, PropertyIdIndexType propertyIdIndex);

void ScopedProperty(OpCode op, RegSlot Value, PropertyIdIndexType propertyIdIndex);

void Slot(OpCode op, RegSlot value, RegSlot instance, int32 slotId);

void Slot(OpCode op, RegSlot value, RegSlot instance, int32 slotId, ProfileId profileId);

void SlotI1(OpCode op, RegSlot value, int32 slotId1);

void SlotI1(OpCode op, RegSlot value, int32 slotId1, ProfileId profileId);

void SlotI2(OpCode op, RegSlot value, int32 slotId1, int32 slotId2);

void SlotI2(OpCode op, RegSlot value, int32 slotId1, int32 slotId2, ProfileId profileId);

void ElementU(OpCode op, RegSlot instance, PropertyIdIndexType propertyIdIndex);

void ElementScopedU(OpCode op, PropertyIdIndexType propertyIdIndex);

void ElementRootU(OpCode op, PropertyIdIndexType propertyIdIndex);

void ElementP(OpCode op, RegSlot value, uint cacheId, bool isCtor = false, bool registerCacheIdForCall = true);

void ElementPIndexed(OpCode op, RegSlot value, uint cacheId, uint32 scopeIndex);

void PatchableRootProperty(OpCode op, RegSlot value, uint cacheId, bool isLoadMethod, bool isStore, bool registerCacheIdForCall = true);

void PatchableProperty(OpCode op, RegSlot value, RegSlot instance, uint cacheId, bool isCtor = false, bool registerCacheIdForCall = true);

void PatchablePropertyWithThisPtr(OpCode op, RegSlot value, RegSlot instance, RegSlot thisInstance, uint cacheId, bool isCtor = false, bool registerCacheIdForCall = true);

void ScopedProperty2(OpCode op, RegSlot Value, PropertyIdIndexType propertyIdIndex, RegSlot R2);

void W1(OpCode op, unsigned short C1);

void Reg1Int2(OpCode op, RegSlot R0, int C1, int C2);

void Reg2Int1(OpCode op, RegSlot R0, RegSlot R1, int C1);

void Unsigned1(OpCode op, uint C1);

void Num3(OpCode op, RegSlot C0, RegSlot C1, RegSlot C2);

template <typename SizePolicy> bool TryWriteReg1(OpCode op, RegSlot R0);

template <typename SizePolicy> bool TryWriteReg2(OpCode op, RegSlot R0, RegSlot R1);

template <typename SizePolicy> bool TryWriteReg2WithICIndex(OpCode op, RegSlot R0, RegSlot R1, uint32 inlineCacheIndex, bool isRootLoad);

template <typename SizePolicy> bool TryWriteReg3(OpCode op, RegSlot R0, RegSlot R1, RegSlot R2);

template <typename SizePolicy> bool TryWriteReg3C(OpCode op, RegSlot R0, RegSlot R1, RegSlot R2, CacheId cacheId);

template <typename SizePolicy> bool TryWriteReg4(OpCode op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3);

template <typename SizePolicy> bool TryWriteReg2B1(OpCode op, RegSlot R0, RegSlot R1, uint8 B2);

template <typename SizePolicy> bool TryWriteReg3B1(OpCode op, RegSlot R0, RegSlot R1, RegSlot R2, uint8 B3);

template <typename SizePolicy> bool TryWriteReg5(OpCode op, RegSlot R0, RegSlot R1, RegSlot R2, RegSlot R3, RegSlot R4);

template <typename SizePolicy> bool TryWriteUnsigned1(OpCode op, uint C1);

template <typename SizePolicy> bool TryWriteArg(OpCode op, ArgSlot arg, RegSlot reg);

template <typename SizePolicy> bool TryWriteArgNoSrc(OpCode op, ArgSlot arg);

template <typename SizePolicy> bool TryWriteBrReg1(OpCode op, ByteCodeLabel labelID, RegSlot R1);

template <typename SizePolicy> bool TryWriteBrReg2(OpCode op, ByteCodeLabel labelID, RegSlot R1, RegSlot R2);

template <typename SizePolicy> bool TryWriteCallI(OpCode op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount);

template <typename SizePolicy> bool TryWriteCallIFlags(OpCode op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, CallFlags callFlags);

template <typename SizePolicy> bool TryWriteCallIWithICIndex(OpCode op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, uint32 inlineCacheIndex, bool isRootLoad);

template <typename SizePolicy> bool TryWriteCallIFlagsWithICIndex(OpCode op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, uint32 inlineCacheIndex, bool isRootLoad, CallFlags callFlags);

template <typename SizePolicy> bool TryWriteCallIExtended(OpCode op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, CallIExtendedOptions options, uint32 spreadArgsOffset);

template <typename SizePolicy> bool TryWriteCallIExtendedWithICIndex(OpCode op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, InlineCacheIndex inlineCacheIndex, bool isRootLoad, CallIExtendedOptions options, uint32 spreadArgsOffset);

template <typename SizePolicy> bool TryWriteCallIExtendedFlags(OpCode op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, CallIExtendedOptions options, uint32 spreadArgsOffset, CallFlags callFlags);

template <typename SizePolicy> bool TryWriteCallIExtendedFlagsWithICIndex(OpCode op, RegSlot returnValueRegister, RegSlot functionRegister, ArgSlot givenArgCount, InlineCacheIndex inlineCacheIndex, bool isRootLoad, CallIExtendedOptions options, uint32 spreadArgsOffset, CallFlags callFlags);

template <typename SizePolicy> bool TryWriteElementI(OpCode op, RegSlot Value, RegSlot Instance, RegSlot Element);

template <typename SizePolicy> bool TryWriteElementUnsigned1(OpCode op, RegSlot Value, RegSlot Instance, uint32 Element);

template <typename SizePolicy> bool TryWriteElementC(OpCode op, RegSlot value, RegSlot instance, PropertyIdIndexType propertyIdIndex);

template <typename SizePolicy> bool TryWriteElementScopedC(OpCode op, RegSlot value, PropertyIdIndexType propertyIdIndex);

template <typename SizePolicy> bool TryWriteElementSlot(OpCode op, RegSlot value, RegSlot instance, int32 slotId);

template <typename SizePolicy> bool TryWriteElementSlotI1(OpCode op, RegSlot value, int32 slotId);

template <typename SizePolicy> bool TryWriteElementSlotI2(OpCode op, RegSlot value, int32 slotId1, int32 slotId2);

template <typename SizePolicy> bool TryWriteElementU(OpCode op, RegSlot instance, PropertyIdIndexType index);

template <typename SizePolicy> bool TryWriteElementScopedU(OpCode op, PropertyIdIndexType index);

template <typename SizePolicy> bool TryWriteElementRootU(OpCode op, PropertyIdIndexType index);

template <typename SizePolicy> bool TryWriteElementRootCP(OpCode op, RegSlot value, uint cacheId, bool isLoadMethod, bool isStore);

template <typename SizePolicy> bool TryWriteElementP(OpCode op, RegSlot value, CacheId cacheId);

template <typename SizePolicy> bool TryWriteElementPIndexed(OpCode op, RegSlot value, uint32 scopeIndex, CacheId cacheId);

template <typename SizePolicy> bool TryWriteElementCP(OpCode op, RegSlot value, RegSlot instance, CacheId cacheId);

template <typename SizePolicy> bool TryWriteElementScopedC2(OpCode op, RegSlot value, PropertyIdIndexType propertyIdIndex, RegSlot instance2);

template <typename SizePolicy> bool TryWriteElementC2(OpCode op, RegSlot value, RegSlot instance, PropertyIdIndexType propertyIdIndex, RegSlot instance2);

template <typename SizePolicy> bool TryWriteClass(OpCode op, RegSlot constructor, RegSlot extends);

template <typename SizePolicy> bool TryWriteReg1Unsigned1(OpCode op, RegSlot R0, uint C1);

template <typename SizePolicy> bool TryWriteReg2Int1(OpCode op, RegSlot R0, RegSlot R1, int C1);

void AuxiliaryContext(OpCode op, RegSlot destinationRegister, const void\* buffer, int byteCount, Js::RegSlot C1);

int Auxiliary(OpCode op, RegSlot destinationRegister, const void\* buffer, int byteCount, int size);

void Auxiliary(OpCode op, RegSlot destinationRegister, uint byteOffset, int size);

int AuxNoReg(OpCode op, const void\* buffer, int byteCount, int size);

void AuxNoReg(OpCode op, uint byteOffset, int size);

int Reg2Aux(OpCode op, RegSlot R0, RegSlot R1, const void\* buffer, int byteCount, int size);

void Reg2Aux(OpCode op, RegSlot R0, RegSlot R1, uint byteOffset, int size);

uint InsertAuxiliaryData(const void\* buffer, uint byteCount);

void InitClass(RegSlot constructor, RegSlot extends = Js::Constants::NoRegister);

void NewFunction(RegSlot destinationRegister, uint index, bool isGenerator);

void NewInnerFunction(RegSlot destinationRegister, uint index, RegSlot environmentRegister, bool isGenerator);

ByteCodeLabel DefineLabel();

void MarkLabel(ByteCodeLabel labelID);

void StartStatement(ParseNode\* node, uint32 tmpRegCount);

void EndStatement(ParseNode\* node);

void StartSubexpression(ParseNode\* node);

void EndSubexpression(ParseNode\* node);

void RecordFrameDisplayRegister(RegSlot slot);

void RecordObjectRegister(RegSlot slot);

uint GetCurrentOffset() const { return (uint)m\_byteCodeData.GetCurrentOffset(); }

DataChunk \* GetCurrentChunk() const { return m\_byteCodeData.GetCurrentChunk(); }

void SetCurrent(uint offset, DataChunk \* chunk) { m\_byteCodeData.SetCurrent(offset, chunk); }

bool ShouldIncrementCallSiteId(OpCode op);

inline void SetCallSiteCount(Js::ProfileId callSiteId) { this->m\_functionWrite->SetProfiledCallSiteCount(callSiteId); }

// Debugger methods.

DebuggerScope\* RecordStartScopeObject(DiagExtraScopesType scopeType, RegSlot scopeLocation = Js::Constants::NoRegister, int\* index = nullptr);

void AddPropertyToDebuggerScope(DebuggerScope\* debuggerScope, RegSlot location, Js::PropertyId propertyId, bool shouldConsumeRegister = true, DebuggerScopePropertyFlags flags = DebuggerScopePropertyFlags\_None, bool isFunctionDeclaration = false);

void RecordEndScopeObject();

DebuggerScope\* GetCurrentDebuggerScope() const { return m\_currentDebuggerScope; }

void UpdateDebuggerPropertyInitializationOffset(Js::DebuggerScope\* currentDebuggerScope, Js::RegSlot location, Js::PropertyId propertyId, bool shouldConsumeRegister = true, int byteCodeOffset = Constants::InvalidOffset, bool isFunctionDeclaration = false);

FunctionBody\* GetFunctionWrite() const { return m\_functionWrite; }

void RecordStatementAdjustment(FunctionBody::StatementAdjustmentType type);

void RecordCrossFrameEntryExitRecord(bool isEnterBlock);

void RecordForInOrOfCollectionScope();

uint EnterLoop(Js::ByteCodeLabel loopEntrance);

void ExitLoop(uint loopId);

bool DoJitLoopBodies() const { return m\_doJitLoopBodies; }

bool DoInterruptProbes() const { return m\_doInterruptProbe; }

static bool DoProfileCallOp(OpCode op)

{

return op >= OpCode::CallI && op <= OpCode::CallIExtendedFlags;

}

bool DoProfileNewScObjectOp(OpCode op)

{

return

!PHASE\_OFF(InlineConstructorsPhase, m\_functionWrite) &&

(op == OpCode::NewScObject || op == OpCode::NewScObjectSpread);

}

bool DoProfileNewScObjArrayOp(OpCode op)

{

return

!PHASE\_OFF(NativeArrayPhase, m\_functionWrite) &&

!m\_functionWrite->GetScriptContext()->IsInDebugMode() &&

(op == OpCode::NewScObjArray || op == OpCode::NewScObjArraySpread);

}

bool DoProfileNewScArrayOp(OpCode op)

{

return

!PHASE\_OFF(NativeArrayPhase, m\_functionWrite) &&

!PHASE\_OFF(NativeNewScArrayPhase, m\_functionWrite) &&

!m\_functionWrite->GetScriptContext()->IsInDebugMode() &&

(op == OpCode::NewScIntArray || op == OpCode::NewScFltArray || op == OpCode::NewScArray);

}

uint ByteCodeWriter::GetTotalSize()

{

return m\_byteCodeData.GetCurrentOffset() + m\_auxiliaryData.GetCurrentOffset() + m\_auxContextData.GetCurrentOffset();

}

#if DBG

bool IsInitialized() const { return isInitialized; }

bool IsInUse() const { return isInUse; }

#endif

};

}

namespace JsUtil

{

template <>

class ValueEntry<Js::ByteCodeWriter::CacheIdUnit>: public BaseValueEntry<Js::ByteCodeWriter::CacheIdUnit>

{

public:

void Clear()

{

this->value = 0;

}

};

};

<?xml version="1.0" encoding="utf-8"?>

<Project DefaultTargets="Build" ToolsVersion="12.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

<Import Condition="'$(ChakraBuildPathImported)'!='true'" Project="$(SolutionDir)Chakra.Build.Paths.props" />

<Import Project="$(BuildConfigPropsPath)Chakra.Build.ProjectConfiguration.props" />

<PropertyGroup Label="Globals">

<TargetName>Chakra.Runtime.ByteCode</TargetName>

<ProjectGuid>{706083F7-6AA4-4558-A153-6352EF9110F5}</ProjectGuid>

<RootNamespace>JS</RootNamespace>

<Keyword>Win32Proj</Keyword>

</PropertyGroup>

<PropertyGroup Label="Configuration">

<ConfigurationType>StaticLibrary</ConfigurationType>

</PropertyGroup>

<Import Project="$(BuildConfigPropsPath)Chakra.Build.Default.props" />

<Import Project="$(VCTargetsPath)\Microsoft.Cpp.Default.props" />

<Import Project="$(VCTargetsPath)\Microsoft.Cpp.props" />

<Import Project="$(BuildConfigPropsPath)Chakra.Build.props" />

<PropertyGroup>

<\_ProjectFileVersion>10.0.30319.1</\_ProjectFileVersion>

</PropertyGroup>

<ItemDefinitionGroup>

<ClCompile>

<AdditionalIncludeDirectories>

$(MSBuildThisFileDirectory)..;

$(MSBuildThisFileDirectory)..\..\Common;

$(MSBuildThisFileDirectory)..\..\Parser;

$(MSBuildThisFileDirectory)..\..\Backend;

%(AdditionalIncludeDirectories)

</AdditionalIncludeDirectories>

<PrecompiledHeader>Use</PrecompiledHeader>

<PrecompiledHeaderFile>RuntimeByteCodePch.h</PrecompiledHeaderFile>

</ClCompile>

</ItemDefinitionGroup>

<ItemGroup>

<ClCompile Include="$(MSBuildThisFileDirectory)AsmJSByteCodeDumper.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)AsmJsByteCodeWriter.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ByteBlock.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ByteCodeDumper.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ByteCodeEmitter.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ByteCodeGenerator.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ByteCodeReader.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ByteCodeWriter.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ByteCodeSerializer.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)FuncInfo.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)Opcodes.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)OpCodeUtil.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)OpCodeUtilAsmJs.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)Scope.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ScopeInfo.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)StatementReader.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)Symbol.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)RuntimeByteCodePch.cpp">

<PrecompiledHeader>Create</PrecompiledHeader>

</ClCompile>

<ClCompile Include="BackEndOpcodeAttr.cpp" />

</ItemGroup>

<ItemGroup>

<ClInclude Include="AsmJSByteCodeDumper.h" />

<ClInclude Include="AsmJsByteCodeWriter.h" />

<ClInclude Include="BackEndOpcodeAttr.h" />

<ClInclude Include="ByteBlock.h" />

<ClInclude Include="ByteCodeAPI.h" />

<ClInclude Include="ByteCodeDumper.h" />

<ClInclude Include="ByteCodeGenerator.h" />

<ClInclude Include="ByteCodeReader.h" />

<ClInclude Include="ByteCodeWriter.h" />

<ClInclude Include="ByteCodeSerializeFlags.h" />

<ClInclude Include="ByteCodeSerializer.h" />

<ClInclude Include="byteCodeCacheReleaseFileVersion.h" />

<ClInclude Include="ExtendedOpCodeList.h" />

<ClInclude Include="ExtendedOpCodeListAsmJs.h" />

<ClInclude Include="FuncInfo.h" />

<ClInclude Include="LayoutTypes.h" />

<ClInclude Include="LayoutTypesAsmJs.h" />

<ClInclude Include="OpCodeList.h" />

<ClInclude Include="OpCodeListAsmJs.h" />

<ClInclude Include="OpCodeUtil.h" />

<ClInclude Include="OpCodeUtilAsmJs.h" />

<ClInclude Include="OpCodes.h" />

<ClInclude Include="OpCodesAsmJs.h" />

<ClInclude Include="OpCodesSimd.h" />

<ClInclude Include="OpLayouts.h" />

<ClInclude Include="OpLayoutsAsmJs.h" />

<ClInclude Include="OpLayoutsCommon.h" />

<ClInclude Include="RuntimeByteCodePch.h" />

<ClInclude Include="Scope.h" />

<ClInclude Include="ScopeInfo.h" />

<ClInclude Include="StatementReader.h" />

<ClInclude Include="Symbol.h" />

</ItemGroup>

<Import Project="$(BuildConfigPropsPath)Chakra.Build.targets" Condition="exists('$(BuildConfigPropsPath)Chakra.Build.targets')" />

<Import Project="$(VCTargetsPath)\Microsoft.Cpp.targets" />

</Project>

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

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

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

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

//

// NOTE: This file is intended to be "#include" multiple times. The call site must define the macro

// "DEF\_OP" to be executed for each entry.

//

#if !defined(DEF\_OP)

#error DEF\_OP must be defined before including this file

#endif

// Define the extended byte code opcode range

#define MACRO\_EXTEND(opcode, layout, attr) DEF\_OP(opcode, layout, attr)

#define MACRO\_EXTEND\_WMS(opcode, layout, attr) DEF\_OP(opcode, layout, OpHasMultiSizeLayout|attr )

#include "OpCodes.h"

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

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

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

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

//

// NOTE: This file is intended to be "#include" multiple times. The call site must define the macro

// "DEF\_OP" to be executed for each entry.

//

#if !defined(DEF\_OP)

#error DEF\_OP must be defined before including this file

#endif

// Define the extended byte code opcode range

#define MACRO\_EXTEND(opcode, layout, attr) DEF\_OP(opcode, layout, attr)

#define MACRO\_EXTEND\_WMS(opcode, layout, attr) DEF\_OP(opcode, layout, OpHasMultiSizeLayout|attr )

#include "OpCodesAsmJs.h"

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

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

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

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

#include "RuntimeByteCodePch.h"

FuncInfo::FuncInfo(

const wchar\_t \*name,

ArenaAllocator \*alloc,

Scope \*paramScope,

Scope \*bodyScope,

ParseNode \*pnode,

Js::ParseableFunctionInfo\* byteCodeFunction)

: alloc(alloc),

varRegsCount(0),

constRegsCount(2),

inArgsCount(0),

innerScopeCount(0),

currentInnerScopeIndex((uint)-1),

firstTmpReg(Js::Constants::NoRegister),

curTmpReg(Js::Constants::NoRegister),

outArgsMaxDepth(0),

outArgsCurrentExpr(0),

#if DBG

outArgsDepth(0),

#endif

name(name),

nullConstantRegister(Js::Constants::NoRegister),

undefinedConstantRegister(Js::Constants::NoRegister),

trueConstantRegister(Js::Constants::NoRegister),

falseConstantRegister(Js::Constants::NoRegister),

thisPointerRegister(Js::Constants::NoRegister),

superRegister(Js::Constants::NoRegister),

superCtorRegister(Js::Constants::NoRegister),

newTargetRegister(Js::Constants::NoRegister),

envRegister(Js::Constants::NoRegister),

frameObjRegister(Js::Constants::NoRegister),

frameSlotsRegister(Js::Constants::NoRegister),

frameDisplayRegister(Js::Constants::NoRegister),

funcObjRegister(Js::Constants::NoRegister),

localClosureReg(Js::Constants::NoRegister),

yieldRegister(Js::Constants::NoRegister),

paramScope(paramScope),

bodyScope(bodyScope),

funcExprScope(nullptr),

root(pnode),

capturedSyms(nullptr),

capturedSymMap(nullptr),

currentChildFunction(nullptr),

currentChildScope(nullptr),

callsEval(false),

childCallsEval(false),

hasArguments(false),

hasHeapArguments(false),

isEventHandler(false),

hasLocalInClosure(false),

hasClosureReference(false),

hasGlobalReference(false),

hasCachedScope(false),

funcExprNameReference(false),

applyEnclosesArgs(false),

escapes(false),

hasDeferredChild(false),

childHasWith(false),

hasLoop(false),

hasEscapedUseNestedFunc(false),

needEnvRegister(false),

hasCapturedThis(false),

staticFuncId(-1),

inlineCacheMap(nullptr),

slotProfileIdMap(alloc),

localPropIdOffset(-1),

sameNameArgsPlaceHolderSlotCount(0),

thisScopeSlot(Js::Constants::NoProperty),

superScopeSlot(Js::Constants::NoProperty),

superCtorScopeSlot(Js::Constants::NoProperty),

newTargetScopeSlot(Js::Constants::NoProperty),

isThisLexicallyCaptured(false),

isSuperLexicallyCaptured(false),

isSuperCtorLexicallyCaptured(false),

isNewTargetLexicallyCaptured(false),

inlineCacheCount(0),

rootObjectLoadInlineCacheCount(0),

rootObjectLoadMethodInlineCacheCount(0),

rootObjectStoreInlineCacheCount(0),

isInstInlineCacheCount(0),

referencedPropertyIdCount(0),

argumentsSymbol(nullptr),

nonUserNonTempRegistersToInitialize(alloc),

constantToRegister(alloc, 17),

stringToRegister(alloc, 17),

doubleConstantToRegister(alloc, 17),

stringTemplateCallsiteRegisterMap(alloc, 17),

targetStatements(alloc)

{

this->byteCodeFunction = byteCodeFunction;

bodyScope->SetFunc(this);

if (paramScope != nullptr)

{

paramScope->SetFunc(this);

}

}

bool FuncInfo::IsGlobalFunction() const

{

return root && root->nop == knopProg;

}

bool FuncInfo::IsDeferred() const

{

return root && root->sxFnc.pnodeBody == nullptr;

}

BOOL FuncInfo::HasSuperReference() const

{

return root->sxFnc.HasSuperReference();

}

BOOL FuncInfo::HasDirectSuper() const

{

return root->sxFnc.HasDirectSuper();

}

BOOL FuncInfo::IsClassMember() const

{

return root->sxFnc.IsClassMember();

}

BOOL FuncInfo::IsLambda() const

{

return root->sxFnc.IsLambda();

}

BOOL FuncInfo::IsClassConstructor() const

{

return root->sxFnc.IsClassConstructor();

}

BOOL FuncInfo::IsBaseClassConstructor() const

{

return root->sxFnc.IsBaseClassConstructor();

}

void FuncInfo::EnsureThisScopeSlot()

{

if (this->thisScopeSlot == Js::Constants::NoRegister)

{

Scope\* scope = this->bodyScope->IsGlobalEvalBlockScope() ? this->GetGlobalEvalBlockScope() : this->bodyScope;

this->thisScopeSlot = scope->AddScopeSlot();

}

}

void FuncInfo::EnsureSuperScopeSlot()

{

if (this->superScopeSlot == Js::Constants::NoRegister)

{

this->superScopeSlot = this->bodyScope->AddScopeSlot();

}

}

void FuncInfo::EnsureSuperCtorScopeSlot()

{

if (this->superCtorScopeSlot == Js::Constants::NoRegister)

{

this->superCtorScopeSlot = this->bodyScope->AddScopeSlot();

}

}

void FuncInfo::EnsureNewTargetScopeSlot()

{

if (this->newTargetScopeSlot == Js::Constants::NoRegister)

{

this->newTargetScopeSlot = this->bodyScope->AddScopeSlot();

}

}

Scope \*

FuncInfo::GetGlobalBlockScope() const

{

Assert(this->IsGlobalFunction());

Scope \* scope = this->root->sxFnc.pnodeScopes->sxBlock.scope;

Assert(scope == nullptr || scope == this->GetBodyScope() || scope->GetEnclosingScope() == this->GetBodyScope());

return scope;

}

Scope \* FuncInfo::GetGlobalEvalBlockScope() const

{

Scope \* globalEvalBlockScope = this->GetGlobalBlockScope();

Assert(globalEvalBlockScope->GetEnclosingScope() == this->GetBodyScope());

Assert(globalEvalBlockScope->GetScopeType() == ScopeType\_GlobalEvalBlock);

return globalEvalBlockScope;

}

uint FuncInfo::FindOrAddReferencedPropertyId(Js::PropertyId propertyId)

{

Assert(propertyId != Js::Constants::NoProperty);

Assert(referencedPropertyIdToMapIndex != nullptr);

if (propertyId < TotalNumberOfBuiltInProperties)

{

return propertyId;

}

uint index;

if (!referencedPropertyIdToMapIndex->TryGetValue(propertyId, &index))

{

index = this->NewReferencedPropertyId();

referencedPropertyIdToMapIndex->Add(propertyId, index);

}

return index + TotalNumberOfBuiltInProperties;

}

uint FuncInfo::FindOrAddRootObjectInlineCacheId(Js::PropertyId propertyId, bool isLoadMethod, bool isStore)

{

Assert(propertyId != Js::Constants::NoProperty);

Assert(!isLoadMethod || !isStore);

uint cacheId;

RootObjectInlineCacheIdMap \* idMap = isStore ? rootObjectStoreInlineCacheMap : isLoadMethod ? rootObjectLoadMethodInlineCacheMap : rootObjectLoadInlineCacheMap;

if (!idMap->TryGetValue(propertyId, &cacheId))

{

cacheId = isStore ? this->NewRootObjectStoreInlineCache() : isLoadMethod ? this->NewRootObjectLoadMethodInlineCache() : this->NewRootObjectLoadInlineCache();

idMap->Add(propertyId, cacheId);

}

return cacheId;

}

#if DBG\_DUMP

void FuncInfo::Dump()

{

Output::Print(L"FuncInfo: CallsEval:%s ChildCallsEval:%s HasArguments:%s HasHeapArguments:%s\n",

IsTrueOrFalse(this->GetCallsEval()),

IsTrueOrFalse(this->GetChildCallsEval()),

IsTrueOrFalse(this->GetHasArguments()),

IsTrueOrFalse(this->GetHasHeapArguments()));

}

#endif

Js::RegSlot FuncInfo::AcquireLoc(ParseNode \*pnode)

{

// Assign a new temp pseudo-register to this expression.

if (pnode->location == Js::Constants::NoRegister)

{

pnode->location = this->AcquireTmpRegister();

}

return pnode->location;

}

Js::RegSlot FuncInfo::AcquireTmpRegister()

{

Assert(this->firstTmpReg != Js::Constants::NoRegister);

// Allocate a new temp pseudo-register, increasing the locals count if necessary.

Assert(this->curTmpReg <= this->varRegsCount && this->curTmpReg >= this->firstTmpReg);

Js::RegSlot tmpReg = this->curTmpReg;

UInt32Math::Inc(this->curTmpReg);

if (this->curTmpReg > this->varRegsCount)

{

this->varRegsCount = this->curTmpReg;

}

return tmpReg;

}

void FuncInfo::ReleaseLoc(ParseNode \*pnode)

{

// Release the temp assigned to this expression so it can be re-used.

if (pnode && pnode->location != Js::Constants::NoRegister)

{

this->ReleaseTmpRegister(pnode->location);

}

}

void FuncInfo::ReleaseLoad(ParseNode \*pnode)

{

// Release any temp register(s) acquired by an EmitLoad.

switch (pnode->nop)

{

case knopDot:

case knopIndex:

case knopCall:

this->ReleaseReference(pnode);

break;

}

this->ReleaseLoc(pnode);

}

void FuncInfo::ReleaseReference(ParseNode \*pnode)

{

// Release any temp(s) assigned to this reference expression so they can be re-used.

switch (pnode->nop)

{

case knopDot:

this->ReleaseLoc(pnode->sxBin.pnode1);

break;

case knopIndex:

this->ReleaseLoc(pnode->sxBin.pnode2);

this->ReleaseLoc(pnode->sxBin.pnode1);

break;

case knopName:

// Do nothing (see EmitReference)

break;

case knopCall:

case knopNew:

// For call/new, we have to release the ArgOut register(s) in reverse order,

// but we have the args in a singly linked list.

// Fortunately, we know that the set we have to release is sequential.

// So find the endpoints of the list and release them in descending order.

if (pnode->sxCall.pnodeArgs)

{

ParseNode \*pnodeArg = pnode->sxCall.pnodeArgs;

Js::RegSlot firstArg = Js::Constants::NoRegister;

Js::RegSlot lastArg = Js::Constants::NoRegister;

if (pnodeArg->nop == knopList)

{

do

{

if (this->IsTmpReg(pnodeArg->sxBin.pnode1->location))

{

lastArg = pnodeArg->sxBin.pnode1->location;

if (firstArg == Js::Constants::NoRegister)

{

firstArg = lastArg;

}

}

pnodeArg = pnodeArg->sxBin.pnode2;

}

while (pnodeArg->nop == knopList);

}

if (this->IsTmpReg(pnodeArg->location))

{

lastArg = pnodeArg->location;

if (firstArg == Js::Constants::NoRegister)

{

// Just one: first and last point to the same node.

firstArg = lastArg;

}

}

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

{

Assert(firstArg != Js::Constants::NoRegister);

Assert(lastArg >= firstArg);

do

{

// Walk down from last to first.

this->ReleaseTmpRegister(lastArg);

} while (lastArg-- > firstArg); // these are unsigned, so (--lastArg >= firstArg) will cause an infinite loop if firstArg is 0 (although that shouldn't happen)

}

}

// Now release the call target.

switch (pnode->sxCall.pnodeTarget->nop)

{

case knopDot:

case knopIndex:

this->ReleaseReference(pnode->sxCall.pnodeTarget);

this->ReleaseLoc(pnode->sxCall.pnodeTarget);

break;

default:

this->ReleaseLoad(pnode->sxCall.pnodeTarget);

break;

}

break;

default:

this->ReleaseLoc(pnode);

break;

}

}

void FuncInfo::ReleaseTmpRegister(Js::RegSlot tmpReg)

{

// Put this reg back on top of the temp stack (if it's a temp).

Assert(tmpReg != Js::Constants::NoRegister);

if (this->IsTmpReg(tmpReg))

{

Assert(tmpReg == this->curTmpReg - 1);

this->curTmpReg--;

}

}

Js::RegSlot FuncInfo::InnerScopeToRegSlot(Scope \*scope) const

{

Js::RegSlot reg = FirstInnerScopeReg();

Assert(reg != Js::Constants::NoRegister);

uint32 index = scope->GetInnerScopeIndex();

return reg + index;

}

Js::RegSlot FuncInfo::FirstInnerScopeReg() const

{

// FunctionBody stores this as a mapped reg. Callers of this function want the pre-mapped value.

Js::RegSlot reg = this->GetParsedFunctionBody()->FirstInnerScopeReg();

Assert(reg != Js::Constants::NoRegister);

return reg - this->constRegsCount;

}

void FuncInfo::SetFirstInnerScopeReg(Js::RegSlot reg)

{

// Just forward to the FunctionBody.

this->GetParsedFunctionBody()->SetFirstInnerScopeReg(reg);

}

void FuncInfo::AddCapturedSym(Symbol \*sym)

{

if (this->capturedSyms == nullptr)

{

this->capturedSyms = Anew(alloc, SymbolTable, alloc);

}

this->capturedSyms->AddNew(sym);

}

void FuncInfo::OnStartVisitFunction(ParseNode \*pnodeFnc)

{

Assert(pnodeFnc->nop == knopFncDecl);

Assert(this->GetCurrentChildFunction() == nullptr);

this->SetCurrentChildFunction(pnodeFnc->sxFnc.funcInfo);

pnodeFnc->sxFnc.funcInfo->SetCurrentChildScope(pnodeFnc->sxFnc.funcInfo->bodyScope);

}

void FuncInfo::OnEndVisitFunction(ParseNode \*pnodeFnc)

{

Assert(pnodeFnc->nop == knopFncDecl);

Assert(this->GetCurrentChildFunction() == pnodeFnc->sxFnc.funcInfo);

pnodeFnc->sxFnc.funcInfo->SetCurrentChildScope(nullptr);

this->SetCurrentChildFunction(nullptr);

}

void FuncInfo::OnStartVisitScope(Scope \*scope)

{

if (scope == nullptr)

{

return;

}

if (scope->GetScopeType() == ScopeType\_Parameter)

{

// If the scopes are unmerged and we are visiting the parameter scope, the child scope will be the function body scope.

Assert(this->GetCurrentChildScope()->GetEnclosingScope() == scope || this->GetCurrentChildScope() == nullptr);

}

else

{

Assert(this->GetCurrentChildScope() == scope->GetEnclosingScope() || this->GetCurrentChildScope() == nullptr);

}

this->SetCurrentChildScope(scope);

}

void FuncInfo::OnEndVisitScope(Scope \*scope)

{

if (scope == nullptr)

{

return;

}

Assert(this->GetCurrentChildScope() == scope);

this->SetCurrentChildScope(scope->GetEnclosingScope());

}

CapturedSymMap \*FuncInfo::EnsureCapturedSymMap()

{

if (this->capturedSymMap == nullptr)

{

this->capturedSymMap = Anew(alloc, CapturedSymMap, alloc);

}

return this->capturedSymMap;

}

void FuncInfo::SetHasMaybeEscapedNestedFunc(DebugOnly(wchar\_t const \* reason))

{

if (PHASE\_TESTTRACE(Js::StackFuncPhase, this->byteCodeFunction) && !hasEscapedUseNestedFunc)

{

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

wchar\_t const \* r = L"";

DebugOnly(r = reason);

Output::Print(L"HasMaybeEscapedNestedFunc (%s): %s (function %s)\n",

r,

this->byteCodeFunction->GetDisplayName(),

this->byteCodeFunction->GetDebugNumberSet(debugStringBuffer));

Output::Flush();

}

hasEscapedUseNestedFunc = true;

}

uint FuncInfo::AcquireInnerScopeIndex()

{

uint index = this->currentInnerScopeIndex;

if (index == (uint)-1)

{

index = 0;

}

else

{

index++;

if (index == (uint)-1)

{

Js::Throw::OutOfMemory();

}

}

if (index == this->innerScopeCount)

{

this->innerScopeCount = index + 1;

}

this->currentInnerScopeIndex = index;

return index;

}

void FuncInfo::ReleaseInnerScopeIndex()

{

uint index = this->currentInnerScopeIndex;

Assert(index != (uint)-1);

if (index == 0)

{

index = (uint)-1;

}

else

{

index--;

}

this->currentInnerScopeIndex = index;

}

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

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

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

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

struct InlineCacheUnit

{

InlineCacheUnit() : loadCacheId((uint)-1), loadMethodCacheId((uint)-1), storeCacheId((uint)-1) {}

union {

struct {

uint loadCacheId;

uint loadMethodCacheId;

uint storeCacheId;

};

struct {

uint cacheId;

};

};

};

typedef JsUtil::BaseDictionary<ParseNode\*, SList<Symbol\*>\*, ArenaAllocator, PowerOf2SizePolicy> CapturedSymMap;

class FuncInfo

{

private:

struct SlotKey

{

Scope\* scope;

Js::PropertyId slot;

};

template<class TSlotKey>

class SlotKeyComparer

{

public:

static bool Equals(TSlotKey key1, TSlotKey key2)

{

return (key1.scope == key2.scope && key1.slot == key2.slot);

}

static int GetHashCode(TSlotKey key)

{

return ::Math::PointerCastToIntegralTruncate<int>(key.scope) | key.slot & ArenaAllocator::ObjectAlignmentMask;

}

};

uint inlineCacheCount;

uint rootObjectLoadInlineCacheCount;

uint rootObjectLoadMethodInlineCacheCount;

uint rootObjectStoreInlineCacheCount;

uint isInstInlineCacheCount;

uint referencedPropertyIdCount;

uint NewInlineCache()

{

AssertMsg(this->inlineCacheCount < (uint)-2, "Inline cache index wrapped around?");

return inlineCacheCount++;

}

uint NewRootObjectLoadInlineCache()

{

AssertMsg(this->rootObjectLoadInlineCacheCount < (uint)-2, "Inline cache index wrapped around?");

return rootObjectLoadInlineCacheCount++;

}

uint NewRootObjectLoadMethodInlineCache()

{

AssertMsg(this->rootObjectLoadMethodInlineCacheCount < (uint)-2, "Inline cache index wrapped around?");

return rootObjectLoadMethodInlineCacheCount++;

}

uint NewRootObjectStoreInlineCache()

{

AssertMsg(this->rootObjectStoreInlineCacheCount < (uint)-2, "Inline cache index wrapped around?");

return rootObjectStoreInlineCacheCount++;

}

uint NewReferencedPropertyId()

{

AssertMsg(this->referencedPropertyIdCount < (uint)-2, "Referenced Property Id index wrapped around?");

return referencedPropertyIdCount++;

}

FuncInfo \*currentChildFunction;

Scope \*currentChildScope;

SymbolTable \*capturedSyms;

CapturedSymMap \*capturedSymMap;

public:

ArenaAllocator \*alloc;

// set in Bind/Assign pass

Js::RegSlot varRegsCount; // number of registers used for non-constants

Js::RegSlot constRegsCount; // number of registers used for constants

Js::ArgSlot inArgsCount; // number of in args (including 'this')

Js::RegSlot outArgsMaxDepth; // max depth of out args stack

Js::RegSlot outArgsCurrentExpr; // max number of out args accumulated in the current nested expression

uint innerScopeCount;

uint currentInnerScopeIndex;

#if DBG

uint32 outArgsDepth; // number of calls nested in a expression

#endif

const wchar\_t \*name; // name of the function

Js::RegSlot nullConstantRegister; // location, if any, of enregistered null constant

Js::RegSlot undefinedConstantRegister; // location, if any, of enregistered undefined constant

Js::RegSlot trueConstantRegister; // location, if any, of enregistered true constant

Js::RegSlot falseConstantRegister; // location, if any, of enregistered false constant

Js::RegSlot thisPointerRegister; // location, if any, of this pointer

Js::RegSlot superRegister; // location, if any, of the super reference

Js::RegSlot superCtorRegister; // location, if any, of the superCtor reference

Js::RegSlot newTargetRegister; // location, if any, of the new.target reference

private:

Js::RegSlot envRegister; // location, if any, of the closure environment

public:

Js::RegSlot frameObjRegister; // location, if any, of the heap-allocated local frame

Js::RegSlot frameSlotsRegister; // location, if any, of the heap-allocated local frame

Js::RegSlot frameDisplayRegister; // location, if any, of the display of nested frames

Js::RegSlot funcObjRegister;

Js::RegSlot localClosureReg;

Js::RegSlot yieldRegister;

Js::RegSlot firstTmpReg;

Js::RegSlot curTmpReg;

int sameNameArgsPlaceHolderSlotCount; // count of place holder slots for same name args

int localPropIdOffset;

Js::RegSlot firstThunkArgReg;

short thunkArgCount;

short staticFuncId;

uint callsEval : 1;

uint childCallsEval : 1;

uint hasArguments : 1;

uint hasHeapArguments : 1;

uint isEventHandler : 1;

uint hasLocalInClosure : 1;

uint hasClosureReference : 1;

uint hasGlobalReference : 1;

uint hasCachedScope : 1;

uint funcExprNameReference : 1;

uint applyEnclosesArgs : 1;

uint escapes : 1;

uint hasDeferredChild : 1; // switch for DeferNested to persist outer scopes

uint childHasWith : 1; // deferNested needs to know if child has with

uint hasLoop : 1;

uint hasEscapedUseNestedFunc : 1;

uint needEnvRegister : 1;

uint hasCapturedThis : 1;

typedef JsUtil::BaseDictionary<uint, Js::RegSlot, ArenaAllocator, PrimeSizePolicy> ConstantRegisterMap;

ConstantRegisterMap constantToRegister; // maps uint constant to register

typedef JsUtil::BaseDictionary<IdentPtr, Js::RegSlot, ArenaAllocator> PidRegisterMap;

PidRegisterMap stringToRegister; // maps string constant to register

typedef JsUtil::BaseDictionary<double,Js::RegSlot, ArenaAllocator, PrimeSizePolicy> DoubleRegisterMap;

DoubleRegisterMap doubleConstantToRegister; // maps double constant to register

typedef JsUtil::BaseDictionary<ParseNodePtr, Js::RegSlot, ArenaAllocator, PowerOf2SizePolicy, Js::StringTemplateCallsiteObjectComparer> StringTemplateCallsiteRegisterMap;

StringTemplateCallsiteRegisterMap stringTemplateCallsiteRegisterMap; // maps string template callsite constant to register

Scope \*paramScope; // top level scope for parameter default values

Scope \*bodyScope; // top level scope of the function body

Scope \*funcExprScope;

ParseNode \*root; // top-level AST for function

Js::ParseableFunctionInfo\* byteCodeFunction; // reference to generated bytecode function (could be defer parsed or actually parsed)

SList<ParseNode\*> targetStatements; // statements that are targets of jumps (break or continue)

Js::ByteCodeLabel singleExit;

typedef SList<InlineCacheUnit> InlineCacheList;

typedef JsUtil::BaseDictionary<Js::PropertyId, InlineCacheList\*, ArenaAllocator, PowerOf2SizePolicy> InlineCacheIdMap;

typedef JsUtil::BaseDictionary<Js::RegSlot, InlineCacheIdMap\*, ArenaAllocator, PowerOf2SizePolicy> InlineCacheMap;

typedef JsUtil::BaseDictionary<Js::PropertyId, uint, ArenaAllocator, PowerOf2SizePolicy> RootObjectInlineCacheIdMap;

typedef JsUtil::BaseDictionary<Js::PropertyId, uint, ArenaAllocator, PowerOf2SizePolicy> ReferencedPropertyIdMap;

RootObjectInlineCacheIdMap \* rootObjectLoadInlineCacheMap;

RootObjectInlineCacheIdMap \* rootObjectLoadMethodInlineCacheMap;

RootObjectInlineCacheIdMap \* rootObjectStoreInlineCacheMap;

InlineCacheMap \* inlineCacheMap;

ReferencedPropertyIdMap \* referencedPropertyIdToMapIndex;

SListBase<uint> valueOfStoreCacheIds;

SListBase<uint> toStringStoreCacheIds;

typedef JsUtil::BaseDictionary<SlotKey, Js::ProfileId, ArenaAllocator, PowerOf2SizePolicy, SlotKeyComparer> SlotProfileIdMap;

SlotProfileIdMap slotProfileIdMap;

Js::PropertyId thisScopeSlot;

Js::PropertyId superScopeSlot;

Js::PropertyId superCtorScopeSlot;

Js::PropertyId newTargetScopeSlot;

bool isThisLexicallyCaptured;

bool isSuperLexicallyCaptured;

bool isSuperCtorLexicallyCaptured;

bool isNewTargetLexicallyCaptured;

Symbol \*argumentsSymbol;

JsUtil::List<Js::RegSlot, ArenaAllocator> nonUserNonTempRegistersToInitialize;

// constRegsCount is set to 2 because R0 is the return register, and R1 is the root object.

FuncInfo(

const wchar\_t \*name,

ArenaAllocator \*alloc,

Scope \*paramScope,

Scope \*bodyScope,

ParseNode \*pnode,

Js::ParseableFunctionInfo\* byteCodeFunction);

uint NewIsInstInlineCache() { return isInstInlineCacheCount++; }

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

uint GetRootObjectLoadInlineCacheCount() const { return this->rootObjectLoadInlineCacheCount; }

uint GetRootObjectLoadMethodInlineCacheCount() const { return this->rootObjectLoadMethodInlineCacheCount; }

uint GetRootObjectStoreInlineCacheCount() const { return this->rootObjectStoreInlineCacheCount; }

uint GetIsInstInlineCacheCount() const { return this->isInstInlineCacheCount; }

uint GetReferencedPropertyIdCount() const { return this->referencedPropertyIdCount; }

void SetFirstTmpReg(Js::RegSlot tmpReg)

{

Assert(this->firstTmpReg == Js::Constants::NoRegister);

Assert(this->curTmpReg == Js::Constants::NoRegister);

this->firstTmpReg = tmpReg;

this->curTmpReg = tmpReg;

}

bool IsTmpReg(Js::RegSlot tmpReg)

{

Assert(this->firstTmpReg != Js::Constants::NoRegister);

return !RegIsConst(tmpReg) && tmpReg >= firstTmpReg;

}

bool RegIsConst(Js::RegSlot reg)

{

// varRegsCount includes the tmp regs, so if reg number is larger than that,

// then it must be in the negative range for const.

return reg >= varRegsCount;

}

Js::RegSlot NextVarRegister()

{

AssertMsg(this->firstTmpReg == Js::Constants::NoRegister, "Shouldn't assign var register after we start allocating temp reg");

Js::RegSlot reg = varRegsCount;

UInt32Math::Inc(varRegsCount);

return REGSLOT\_TO\_VARREG(reg);

}

Js::RegSlot NextConstRegister()

{

AssertMsg(this->firstTmpReg == Js::Constants::NoRegister, "Shouldn't assign var register after we start allocating temp reg");

Js::RegSlot reg = constRegsCount;

UInt32Math::Inc(constRegsCount);

return REGSLOT\_TO\_CONSTREG(reg);

}

Js::RegSlot RegCount() const

{

return constRegsCount + varRegsCount;

}

uint InnerScopeCount() const { return innerScopeCount; }

uint CurrentInnerScopeIndex() const { return currentInnerScopeIndex; }

uint AcquireInnerScopeIndex();

void ReleaseInnerScopeIndex();

bool GetApplyEnclosesArgs() const { return applyEnclosesArgs; }

void SetApplyEnclosesArgs(bool b) { applyEnclosesArgs=b; }

bool IsGlobalFunction() const;

// Fake global ->

// 1) new Function code's global code

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

bool IsFakeGlobalFunction(ulong flags) const {

return IsGlobalFunction() && !(flags & fscrGlobalCode);

}

Scope \*GetBodyScope() const {

return bodyScope;

}

Scope \*GetParamScope() const {

return paramScope;

}

Scope \*GetTopLevelScope() const {

// Top level scope will be the same for knopProg and knopFncDecl.

return paramScope;

}

Scope\* GetFuncExprScope() const {

return funcExprScope;

}

void SetFuncExprScope(Scope\* funcExprScope) {

this->funcExprScope = funcExprScope;

}

Symbol \*GetArgumentsSymbol() const

{

return argumentsSymbol;

}

void SetArgumentsSymbol(Symbol \*sym)

{

Assert(argumentsSymbol == nullptr || argumentsSymbol == sym);

argumentsSymbol = sym;

}

bool GetCallsEval() const {

return callsEval;

}

void SetCallsEval(bool does) {

callsEval = does;

}

bool GetHasArguments() const {

return hasArguments;

}

void SetHasArguments(bool has) {

hasArguments = has;

}

bool GetHasHeapArguments() const

{

return hasHeapArguments;

}

void SetHasHeapArguments(bool has, bool optArgInBackend = false)

{

hasHeapArguments = has;

byteCodeFunction->SetDoBackendArgumentsOptimization(optArgInBackend);

}

bool GetIsEventHandler() const {

return isEventHandler;

}

void SetIsEventHandler(bool is) {

isEventHandler = is;

}

bool GetChildCallsEval() const {

return childCallsEval;

}

void SetChildCallsEval(bool does) {

childCallsEval = does;

}

bool GetHasLocalInClosure() const {

return hasLocalInClosure;

}

void SetHasLocalInClosure(bool has) {

hasLocalInClosure = has;

}

bool GetHasClosureReference() const {

return hasClosureReference;

}

void SetHasCachedScope(bool has) {

hasCachedScope = has;

}

bool GetHasCachedScope() const {

return hasCachedScope;

}

void SetFuncExprNameReference(bool has) {

funcExprNameReference = has;

}

bool GetFuncExprNameReference() const {

return funcExprNameReference;

}

void SetHasClosureReference(bool has) {

hasClosureReference = has;

}

bool GetHasGlobalRef() const {

return hasGlobalReference;

}

void SetHasGlobalRef(bool has) {

hasGlobalReference = has;

}

bool GetIsStrictMode() const {

return this->byteCodeFunction->GetIsStrictMode();

}

bool Escapes() const {

return escapes;

}

void SetEscapes(bool does) {

escapes = does;

}

bool HasMaybeEscapedNestedFunc() const {

return hasEscapedUseNestedFunc;

}

void SetHasMaybeEscapedNestedFunc(DebugOnly(wchar\_t const \* reason));

bool IsDeferred() const;

bool IsRestored()

{

// FuncInfo are from RestoredScopeInfo

return root == nullptr;

}

bool HasDeferredChild() const {

return hasDeferredChild;

}

void SetHasDeferredChild() {

hasDeferredChild = true;

}

Js::FunctionBody\* GetParsedFunctionBody() const

{

AssertMsg(this->byteCodeFunction->IsFunctionParsed(), "Function must be parsed in order to call this method");

Assert(!IsDeferred());

return this->byteCodeFunction->GetFunctionBody();

}

bool ChildHasWith() const {

return childHasWith;

}

void SetChildHasWith() {

childHasWith = true;

}

bool HasCapturedThis() const {

return hasCapturedThis;

}

void SetHasCapturedThis() {

hasCapturedThis = true;

}

BOOL HasSuperReference() const;

BOOL HasDirectSuper() const;

BOOL IsClassMember() const;

BOOL IsLambda() const;

BOOL IsClassConstructor() const;

BOOL IsBaseClassConstructor() const;

void RemoveTargetStmt(ParseNode\* pnodeStmt) {

targetStatements.Remove(pnodeStmt);

}

void AddTargetStmt(ParseNode \*pnodeStmt) {

targetStatements.Prepend(pnodeStmt);

}

Js::RegSlot LookupDouble(double d) {

return doubleConstantToRegister.Lookup(d,Js::Constants::NoRegister);

}

bool TryGetDoubleLoc(double d, Js::RegSlot \*loc) {

Js::RegSlot ret=LookupDouble(d);

\*loc=ret;

return(ret!=Js::Constants::NoRegister);

}

void AddDoubleConstant(double d, Js::RegSlot location) {

doubleConstantToRegister.Item(d,location);

}

bool NeedEnvRegister() const { return this->needEnvRegister; }

void SetNeedEnvRegister() { this->needEnvRegister = true; };

Js::RegSlot GetEnvRegister() const

{

Assert(this->envRegister != Js::Constants::NoRegister);

return this->envRegister;

}

Js::RegSlot AssignEnvRegister(bool constReg)

{

Assert(needEnvRegister);

Assert(this->envRegister == Js::Constants::NoRegister);

Js::RegSlot reg = constReg? NextConstRegister() : NextVarRegister();

this->envRegister = reg;

return reg;

}

Js::RegSlot AssignThisRegister()

{

if (this->thisPointerRegister == Js::Constants::NoRegister)

{

this->thisPointerRegister = NextVarRegister();

}

return this->thisPointerRegister;

}

Js::RegSlot AssignSuperRegister()

{

if (this->superRegister == Js::Constants::NoRegister)

{

this->superRegister = NextVarRegister();

}

return this->superRegister;

}

Js::RegSlot AssignSuperCtorRegister()

{

if (this->superCtorRegister == Js::Constants::NoRegister)

{

this->superCtorRegister = NextVarRegister();

}

return this->superCtorRegister;

}

Js::RegSlot AssignNewTargetRegister()

{

if (this->newTargetRegister == Js::Constants::NoRegister)

{

this->newTargetRegister = NextVarRegister();

}

return this->newTargetRegister;

}

Js::RegSlot AssignNullConstRegister()

{

if (this->nullConstantRegister == Js::Constants::NoRegister)

{

this->nullConstantRegister = NextConstRegister();

}

return this->nullConstantRegister;

}

Js::RegSlot AssignUndefinedConstRegister()

{

if (this->undefinedConstantRegister == Js::Constants::NoRegister)

{

this->undefinedConstantRegister = NextConstRegister();

}

return this->undefinedConstantRegister;

}

Js::RegSlot AssignTrueConstRegister()

{

if (this->trueConstantRegister == Js::Constants::NoRegister)

{

this->trueConstantRegister = NextConstRegister();

}

return this->trueConstantRegister;

}

Js::RegSlot AssignFalseConstRegister()

{

if (this->falseConstantRegister == Js::Constants::NoRegister)

{

this->falseConstantRegister = NextConstRegister();

}

return this->falseConstantRegister;

}

Js::RegSlot AssignYieldRegister()

{

AssertMsg(this->yieldRegister == Js::Constants::NoRegister, "yield register should only be assigned once by FinalizeRegisters()");

this->yieldRegister = NextVarRegister();

return this->yieldRegister;

}

Js::RegSlot GetLocalScopeSlotsReg()

{

return this->localClosureReg;

}

Js::RegSlot GetLocalFrameDisplayReg()

{

return this->localClosureReg + 1;

}

Js::RegSlot InnerScopeToRegSlot(Scope \*scope) const;

Js::RegSlot FirstInnerScopeReg() const;

void SetFirstInnerScopeReg(Js::RegSlot reg);

void StartRecordingOutArgs(unsigned int argCount)

{

#if DBG

outArgsDepth++;

#endif

// We should have checked for argCount overflow already

Assert(argCount == (Js::ArgSlot)argCount);

// Add one for the space to save the m\_outParams pointer in InterpreterStackFrame::PushOut

unsigned int outArgsCount = argCount + 1;

outArgsCurrentExpr += (Js::ArgSlot)outArgsCount;

// Check for overflow

if ((Js::ArgSlot)outArgsCount != outArgsCount || outArgsCurrentExpr < outArgsCount)

{

Js::Throw::OutOfMemory();

}

outArgsMaxDepth = max(outArgsMaxDepth, outArgsCurrentExpr);

}

void EndRecordingOutArgs(Js::ArgSlot argCount)

{

AssertMsg(outArgsDepth > 0, "mismatched Start and End");

Assert(outArgsCurrentExpr >= argCount);

#if DBG

outArgsDepth--;

#endif

// Add one to pop the space to save the m\_outParams pointer

outArgsCurrentExpr -= (argCount + 1);

Assert(outArgsDepth != 0 || outArgsCurrentExpr == 0);

}

Js::RegSlot AcquireLoc(ParseNode \*pnode);

Js::RegSlot AcquireTmpRegister();

void ReleaseLoc(ParseNode \*pnode);

void ReleaseReference(ParseNode \*pnode);

void ReleaseLoad(ParseNode \*pnode);

void ReleaseTmpRegister(Js::RegSlot tmpReg);

uint FindOrAddReferencedPropertyId(Js::PropertyId propertyId);

uint FindOrAddRootObjectInlineCacheId(Js::PropertyId propertyId, bool isLoadMethod, bool isStore);

uint FindOrAddInlineCacheId(Js::RegSlot instanceSlot, Js::PropertyId propertySlot, bool isLoadMethod, bool isStore)

{

Assert(instanceSlot != Js::Constants::NoRegister);

Assert(propertySlot != Js::Constants::NoProperty);

Assert(!isLoadMethod || !isStore);

InlineCacheIdMap \*properties;

uint cacheId;

if (isStore)

{

// ... = foo.toString;

// foo.toString = ...;

// We need a new cache here to ensure SetProperty() is called, which will set the side-effect bit

// on the scriptContext.

switch (propertySlot)

{

case Js::PropertyIds::valueOf:

cacheId = this->NewInlineCache();

valueOfStoreCacheIds.Prepend(alloc, cacheId);

return cacheId;

case Js::PropertyIds::toString:

cacheId = this->NewInlineCache();

toStringStoreCacheIds.Prepend(alloc, cacheId);

return cacheId;

};

}

if (!inlineCacheMap->TryGetValue(instanceSlot, &properties))

{

properties = Anew(alloc, InlineCacheIdMap, alloc, 17);

inlineCacheMap->Add(instanceSlot, properties);

}

InlineCacheList\* cacheList;

if (!properties->TryGetValue(propertySlot, &cacheList))

{

cacheList = Anew(alloc, InlineCacheList, alloc);

properties->Add(propertySlot, cacheList);

}

// If we share inline caches we should never have more than one entry in the list.

Assert(Js::FunctionBody::ShouldShareInlineCaches() || cacheList->Count() <= 1);

InlineCacheUnit cacheIdUnit;

if (Js::FunctionBody::ShouldShareInlineCaches() && !cacheList->Empty())

{

cacheIdUnit = cacheList->Head();

if (isLoadMethod)

{

if (cacheIdUnit.loadMethodCacheId == (uint)-1)

{

cacheIdUnit.loadMethodCacheId = this->NewInlineCache();

}

cacheId = cacheIdUnit.loadMethodCacheId;

}

else if (isStore)

{

if (cacheIdUnit.storeCacheId == (uint)-1)

{

cacheIdUnit.storeCacheId = this->NewInlineCache();

}

cacheId = cacheIdUnit.storeCacheId;

}

else

{

if (cacheIdUnit.loadCacheId == (uint)-1)

{

cacheIdUnit.loadCacheId = this->NewInlineCache();

}

cacheId = cacheIdUnit.loadCacheId;

}

cacheList->Head() = cacheIdUnit;

}

else

{

cacheId = this->NewInlineCache();

if (Js::FunctionBody::ShouldShareInlineCaches())

{

if (isLoadMethod)

{

cacheIdUnit.loadCacheId = (uint)-1;

cacheIdUnit.loadMethodCacheId = cacheId;

cacheIdUnit.storeCacheId = (uint)-1;

}

else if (isStore)

{

cacheIdUnit.loadCacheId = (uint)-1;

cacheIdUnit.loadMethodCacheId = (uint)-1;

cacheIdUnit.storeCacheId = cacheId;

}

else

{

cacheIdUnit.loadCacheId = cacheId;

cacheIdUnit.loadMethodCacheId = (uint)-1;

cacheIdUnit.storeCacheId = (uint)-1;

}

}

else

{

cacheIdUnit.cacheId = cacheId;

}

cacheList->Prepend(cacheIdUnit);

}

return cacheId;

}

Js::ProfileId FindOrAddSlotProfileId(Scope\* scope, Js::PropertyId propertyId)

{

SlotKey key;

key.scope = scope;

key.slot = propertyId;

Js::ProfileId profileId = Js::Constants::NoProfileId;

if (!this->slotProfileIdMap.TryGetValue(key, &profileId))

{

Assert(this->byteCodeFunction->IsFunctionParsed());

if (this->byteCodeFunction->GetFunctionBody()->AllocProfiledSlotId(&profileId))

{

this->slotProfileIdMap.Add(key, profileId);

}

}

return profileId;

}

void EnsureThisScopeSlot();

void EnsureSuperScopeSlot();

void EnsureSuperCtorScopeSlot();

void EnsureNewTargetScopeSlot();

void SetIsThisLexicallyCaptured()

{

this->isThisLexicallyCaptured = true;

}

void SetIsSuperLexicallyCaptured()

{

this->isSuperLexicallyCaptured = true;

}

void SetIsSuperCtorLexicallyCaptured()

{

this->isSuperCtorLexicallyCaptured = true;

}

void SetIsNewTargetLexicallyCaptured()

{

this->isNewTargetLexicallyCaptured = true;

}

Scope \* GetGlobalBlockScope() const;

Scope \* GetGlobalEvalBlockScope() const;

FuncInfo \*GetCurrentChildFunction() const

{

return this->currentChildFunction;

}

void SetCurrentChildFunction(FuncInfo \*funcInfo)

{

this->currentChildFunction = funcInfo;

}

Scope \*GetCurrentChildScope() const

{

return this->currentChildScope;

}

void SetCurrentChildScope(Scope \*scope)

{

this->currentChildScope = scope;

}

SymbolTable \*GetCapturedSyms() const { return capturedSyms; }

void OnStartVisitFunction(ParseNode \*pnodeFnc);

void OnEndVisitFunction(ParseNode \*pnodeFnc);

void OnStartVisitScope(Scope \*scope);

void OnEndVisitScope(Scope \*scope);

void AddCapturedSym(Symbol \*sym);

CapturedSymMap \*EnsureCapturedSymMap();

#if DBG\_DUMP

void Dump();

#endif

};

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

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

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

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

//

// NOTE: This file is intended to be "#include" multiple times. The call site must define the macros

// "LAYOUT\_TYPE", etc., to be executed for each entry.

//

#ifndef LAYOUT\_TYPE

#define LAYOUT\_TYPE(layout)

#endif

#ifndef LAYOUT\_TYPE\_WMS

#define LAYOUT\_TYPE\_WMS(layout) \

LAYOUT\_TYPE(layout##\_Small) \

LAYOUT\_TYPE(layout##\_Medium) \

LAYOUT\_TYPE(layout##\_Large)

#endif

#ifndef LAYOUT\_TYPE\_PROFILED

#define LAYOUT\_TYPE\_PROFILED(layout) \

LAYOUT\_TYPE(layout) \

LAYOUT\_TYPE(Profiled##layout)

#endif

#ifndef LAYOUT\_TYPE\_PROFILED2

#define LAYOUT\_TYPE\_PROFILED2(layout) \

LAYOUT\_TYPE\_PROFILED(layout) \

LAYOUT\_TYPE(Profiled2##layout)

#endif

#ifndef LAYOUT\_TYPE\_PROFILED\_WMS

#define LAYOUT\_TYPE\_PROFILED\_WMS(layout) \

LAYOUT\_TYPE\_WMS(layout) \

LAYOUT\_TYPE\_WMS(Profiled##layout)

#endif

#ifndef LAYOUT\_TYPE\_PROFILED2\_WMS

#define LAYOUT\_TYPE\_PROFILED2\_WMS(layout) \

LAYOUT\_TYPE\_PROFILED\_WMS(layout) \

LAYOUT\_TYPE\_WMS(Profiled2##layout)

#endif

LAYOUT\_TYPE (Empty)

LAYOUT\_TYPE\_WMS (Reg1)

LAYOUT\_TYPE\_PROFILED\_WMS (Reg2)

LAYOUT\_TYPE\_PROFILED\_WMS (Reg2WithICIndex)

LAYOUT\_TYPE\_PROFILED\_WMS (Reg3)

LAYOUT\_TYPE\_WMS (Reg4)

LAYOUT\_TYPE\_PROFILED\_WMS (Reg1Unsigned1)

LAYOUT\_TYPE\_WMS (Reg2B1)

LAYOUT\_TYPE\_WMS (Reg3B1)

LAYOUT\_TYPE\_WMS (Reg3C)

LAYOUT\_TYPE\_PROFILED\_WMS (Arg)

LAYOUT\_TYPE\_WMS (ArgNoSrc)

#ifdef BYTECODE\_BRANCH\_ISLAND

LAYOUT\_TYPE (BrLong)

#endif

LAYOUT\_TYPE (Br)

LAYOUT\_TYPE\_WMS (BrReg1)

LAYOUT\_TYPE\_WMS (BrReg2)

LAYOUT\_TYPE (BrS)

LAYOUT\_TYPE (BrProperty)

LAYOUT\_TYPE (BrLocalProperty)

LAYOUT\_TYPE (BrEnvProperty)

LAYOUT\_TYPE (StartCall)

LAYOUT\_TYPE\_PROFILED2\_WMS (CallI)

LAYOUT\_TYPE\_PROFILED\_WMS (CallIFlags)

LAYOUT\_TYPE\_PROFILED\_WMS (CallIWithICIndex)

LAYOUT\_TYPE\_PROFILED\_WMS (CallIFlagsWithICIndex)

LAYOUT\_TYPE\_PROFILED2\_WMS (CallIExtended)

LAYOUT\_TYPE\_PROFILED\_WMS (CallIExtendedFlags)

LAYOUT\_TYPE\_PROFILED\_WMS (CallIExtendedWithICIndex)

LAYOUT\_TYPE\_PROFILED\_WMS (CallIExtendedFlagsWithICIndex)

LAYOUT\_TYPE\_WMS (Class)

LAYOUT\_TYPE\_PROFILED\_WMS (ElementI)

LAYOUT\_TYPE\_WMS (ElementUnsigned1)

LAYOUT\_TYPE\_WMS (ElementC)

LAYOUT\_TYPE\_WMS (ElementScopedC)

LAYOUT\_TYPE\_WMS (ElementCP)

LAYOUT\_TYPE\_WMS (ElementP)

LAYOUT\_TYPE\_WMS (ElementPIndexed)

LAYOUT\_TYPE\_WMS (ElementRootCP)

LAYOUT\_TYPE\_WMS (ElementC2)

LAYOUT\_TYPE\_WMS (ElementScopedC2)

LAYOUT\_TYPE\_PROFILED\_WMS (ElementSlot)

LAYOUT\_TYPE\_PROFILED\_WMS (ElementSlotI1)

LAYOUT\_TYPE\_PROFILED\_WMS (ElementSlotI2)

LAYOUT\_TYPE\_WMS (ElementU)

LAYOUT\_TYPE\_WMS (ElementScopedU)

LAYOUT\_TYPE\_WMS (ElementRootU)

LAYOUT\_TYPE (W1)

LAYOUT\_TYPE (Reg1Int2)

LAYOUT\_TYPE\_WMS (Reg2Int1)

LAYOUT\_TYPE (AuxNoReg)

LAYOUT\_TYPE\_PROFILED (Auxiliary)

LAYOUT\_TYPE (Reg2Aux)

LAYOUT\_TYPE\_WMS (Reg5)

LAYOUT\_TYPE\_WMS (Unsigned1)

#undef LAYOUT\_TYPE

#undef LAYOUT\_TYPE\_WMS

#undef LAYOUT\_TYPE\_PROFILED2\_WMS

#undef LAYOUT\_TYPE\_PROFILED\_WMS

#undef LAYOUT\_TYPE\_PROFILED

#undef LAYOUT\_TYPE\_PROFILED2

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

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

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

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

//

// NOTE: This file is intended to be "#include" multiple times. The call site must define the macros

// "LAYOUT\_TYPE", etc., to be executed for each entry.

//

#ifndef TEMP\_DISABLE\_ASMJS

#ifndef LAYOUT\_TYPE

#define LAYOUT\_TYPE(layout)

#endif

#ifndef LAYOUT\_TYPE\_WMS

#define LAYOUT\_TYPE\_WMS(layout) \

LAYOUT\_TYPE(layout##\_Small) \

LAYOUT\_TYPE(layout##\_Medium) \

LAYOUT\_TYPE(layout##\_Large)

#endif

// FE for frontend only layout

#ifdef EXCLUDE\_FRONTEND\_LAYOUT

#ifndef LAYOUT\_TYPE\_WMS\_FE

#define LAYOUT\_TYPE\_WMS\_FE(...)

#endif

#else

#ifndef LAYOUT\_TYPE\_WMS\_FE

#define LAYOUT\_TYPE\_WMS\_FE LAYOUT\_TYPE\_WMS

#endif

#endif

// For duplicates layout from LayoutTypes.h

#ifdef EXCLUDE\_DUP\_LAYOUT

#define LAYOUT\_TYPE\_DUP(...)

#define LAYOUT\_TYPE\_WMS\_DUP(...)

#else

#define LAYOUT\_TYPE\_DUP LAYOUT\_TYPE

#define LAYOUT\_TYPE\_WMS\_DUP LAYOUT\_TYPE\_WMS

#endif

// These layout are already defined in LayoutTypes.h

// We redeclare them here to keep the same layout and use them

// in other contexts.

LAYOUT\_TYPE\_WMS\_DUP ( ElementSlot )

LAYOUT\_TYPE\_DUP ( StartCall )

LAYOUT\_TYPE\_DUP ( Empty )

LAYOUT\_TYPE\_WMS ( AsmTypedArr )

LAYOUT\_TYPE\_WMS ( AsmCall )

LAYOUT\_TYPE ( AsmBr )

LAYOUT\_TYPE\_WMS ( AsmReg1 ) // Generic layout with 1 RegSlot

LAYOUT\_TYPE\_WMS\_FE ( AsmReg2 ) // Generic layout with 2 RegSlot

LAYOUT\_TYPE\_WMS\_FE ( AsmReg3 ) // Generic layout with 3 RegSlot

LAYOUT\_TYPE\_WMS\_FE ( AsmReg4 ) // Generic layout with 4 RegSlot

LAYOUT\_TYPE\_WMS\_FE ( AsmReg5 ) // Generic layout with 5 RegSlot

LAYOUT\_TYPE\_WMS\_FE ( AsmReg6 ) // Generic layout with 6 RegSlot

LAYOUT\_TYPE\_WMS\_FE ( AsmReg7 ) // Generic layout with 7 RegSlot

LAYOUT\_TYPE\_WMS\_FE ( AsmReg2IntConst1 ) // Generic layout with 2 RegSlots and 1 Int Constant

LAYOUT\_TYPE\_WMS ( Int1Double1 ) // 1 int register and 1 double register

LAYOUT\_TYPE\_WMS ( Int1Float1 ) // 1 int register and 1 float register

LAYOUT\_TYPE\_WMS ( Double1Int1 ) // 1 double register and 1 int register

LAYOUT\_TYPE\_WMS ( Double1Float1 ) // 1 double register and 1 float register

LAYOUT\_TYPE\_WMS ( Double1Reg1 ) // 1 double register and 1 var register

LAYOUT\_TYPE\_WMS ( Float1Reg1 ) // 1 double register and 1 var register

LAYOUT\_TYPE\_WMS ( Int1Reg1 ) // 1 int register and 1 var register

LAYOUT\_TYPE\_WMS ( Reg1Double1 ) // 1 var register and 1 double register

LAYOUT\_TYPE\_WMS ( Reg1Float1 ) // 1 var register and 1 Float register

LAYOUT\_TYPE\_WMS ( Reg1Int1 ) // 1 var register and 1 int register

LAYOUT\_TYPE\_WMS ( Int1Const1 ) // 1 int register and 1 const int value

LAYOUT\_TYPE\_WMS ( Int1Double2 ) // 1 int register and 2 double register ( double comparisons )

LAYOUT\_TYPE\_WMS ( Int1Float2 ) // 1 int register and 2 float register ( float comparisons )

LAYOUT\_TYPE\_WMS ( Int2 ) // 2 int register

LAYOUT\_TYPE\_WMS ( Int3 ) // 3 int register

LAYOUT\_TYPE\_WMS ( Double2 ) // 2 double register

LAYOUT\_TYPE\_WMS ( Float2 ) // 2 float register

LAYOUT\_TYPE\_WMS ( Float3 ) // 3 float register

LAYOUT\_TYPE\_WMS ( Float1Double1 ) // 2 double register

LAYOUT\_TYPE\_WMS ( Float1Int1 ) // 2 double register

LAYOUT\_TYPE\_WMS ( Double3 ) // 3 double register

LAYOUT\_TYPE\_WMS ( BrInt1 ) // Conditional branching with 1 int

LAYOUT\_TYPE\_WMS ( BrInt2 ) // Conditional branching with 2 int

LAYOUT\_TYPE\_WMS ( AsmUnsigned1 ) // Conditional branching with 2 int

// Float32x4

LAYOUT\_TYPE\_WMS ( Float32x4\_2 )

LAYOUT\_TYPE\_WMS ( Float32x4\_3 )

LAYOUT\_TYPE\_WMS ( Float32x4\_4 )

LAYOUT\_TYPE\_WMS ( Float32x4\_1Float4 )

LAYOUT\_TYPE\_WMS ( Float32x4\_2Int4 )

LAYOUT\_TYPE\_WMS ( Float32x4\_3Int4 )

LAYOUT\_TYPE\_WMS ( Float32x4\_1Float1 )

LAYOUT\_TYPE\_WMS ( Float32x4\_2Float1 )

LAYOUT\_TYPE\_WMS ( Float32x4\_1Float64x2\_1 )

LAYOUT\_TYPE\_WMS ( Float32x4\_1Int32x4\_1 )

LAYOUT\_TYPE\_WMS ( Float32x4\_1Int32x4\_1Float32x4\_2 )

LAYOUT\_TYPE\_WMS ( Reg1Float32x4\_1 )

LAYOUT\_TYPE\_WMS ( Float1Float32x4\_1Int1 )

LAYOUT\_TYPE\_WMS ( Float32x4\_2Int1Float1 )

LAYOUT\_TYPE\_WMS ( Int1Float32x4\_1)

// Int32x4

LAYOUT\_TYPE\_WMS ( Int32x4\_2)

LAYOUT\_TYPE\_WMS ( Int32x4\_3)

LAYOUT\_TYPE\_WMS ( Int32x4\_4)

LAYOUT\_TYPE\_WMS ( Int32x4\_1Int4 )

LAYOUT\_TYPE\_WMS ( Int32x4\_2Int4 )

LAYOUT\_TYPE\_WMS ( Int32x4\_3Int4 )

LAYOUT\_TYPE\_WMS ( Int32x4\_1Int1 )

LAYOUT\_TYPE\_WMS ( Int32x4\_2Int1 )

LAYOUT\_TYPE\_WMS ( Reg1Int32x4\_1 )

LAYOUT\_TYPE\_WMS ( Int32x4\_1Float32x4\_1 )

LAYOUT\_TYPE\_WMS ( Int32x4\_1Float64x2\_1 )

LAYOUT\_TYPE\_WMS ( Int1Int32x4\_1)

LAYOUT\_TYPE\_WMS ( Int1Int32x4\_1Int1 )

LAYOUT\_TYPE\_WMS ( Int32x4\_2Int2 )

// Float64x2

LAYOUT\_TYPE\_WMS ( Float64x2\_2 )

LAYOUT\_TYPE\_WMS ( Float64x2\_3 )

LAYOUT\_TYPE\_WMS ( Float64x2\_4 )

LAYOUT\_TYPE\_WMS ( Float64x2\_1Double2 )

LAYOUT\_TYPE\_WMS ( Float64x2\_1Double1 )

LAYOUT\_TYPE\_WMS ( Float64x2\_2Double1 )

LAYOUT\_TYPE\_WMS ( Float64x2\_2Int2 )

LAYOUT\_TYPE\_WMS ( Float64x2\_3Int2 )

LAYOUT\_TYPE\_WMS ( Float64x2\_1Float32x4\_1 )

LAYOUT\_TYPE\_WMS ( Float64x2\_1Int32x4\_1 )

LAYOUT\_TYPE\_WMS ( Float64x2\_1Int32x4\_1Float64x2\_2 )

LAYOUT\_TYPE\_WMS ( Reg1Float64x2\_1 )

LAYOUT\_TYPE\_WMS ( Int1Float64x2\_1)

LAYOUT\_TYPE\_WMS ( AsmSimdTypedArr )

#undef LAYOUT\_TYPE\_DUP

#undef LAYOUT\_TYPE\_WMS\_DUP

#undef LAYOUT\_TYPE

#undef LAYOUT\_TYPE\_WMS

#undef EXCLUDE\_DUP\_LAYOUT

#undef LAYOUT\_TYPE\_WMS\_FE

#undef EXCLUDE\_FRONTEND\_LAYOUT

#endif

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

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

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

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

//

// NOTE: This file is intended to be "#include" multiple times. The call site must define the macro

// "DEF\_OP" to be executed for each entry.

//

#if !defined(DEF\_OP)

#error DEF\_OP must be defined before including this file

#endif

// Define the basic byte code opcode range

#define MACRO(opcode, layout, attr) DEF\_OP(opcode, layout, attr)

#define MACRO\_WMS(opcode, layout, attr) DEF\_OP(opcode, layout, OpHasMultiSizeLayout|attr )

#include "OpCodes.h"

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

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

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

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

//

// NOTE: This file is intended to be "#include" multiple times. The call site must define the macro

// "DEF\_OP" to be executed for each entry.

//

#if !defined(DEF\_OP)

#error DEF\_OP must be defined before including this file

#endif

// Define the basic byte code opcode range

#define MACRO(opcode, layout, attr) DEF\_OP(opcode, layout, attr)

#define MACRO\_WMS(opcode, layout, attr) DEF\_OP(opcode, layout, OpHasMultiSizeLayout|attr )

#include "OpCodesAsmJs.h"

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

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

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

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

#include "RuntimeByteCodePch.h"

// We only have one extended range so maximum number of opcode is only 512 unless we add more extended ranges.

CompileAssert((uint)Js::OpCode::ByteCodeLast < 512);

// Make sure all basic opcode with no one byte layout fits in a byte.

#define MACRO(opcode, layout, attr) CompileAssert((uint)Js::OpCode::opcode <= BYTE\_MAX);

#define MACRO\_WMS(opcode, layout, attr) CompileAssert((uint)Js::OpCode::opcode <= BYTE\_MAX);

// Make sure all extended opcode needs two bytes.

#define MACRO\_EXTEND(opcode, layout, attr) CompileAssert((uint)Js::OpCode::opcode > BYTE\_MAX);

#define MACRO\_EXTEND\_WMS(opcode, layout, attr) CompileAssert((uint)Js::OpCode::opcode > BYTE\_MAX);

#include "OpCodes.h"

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

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

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

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

// Default all macro to nothing

#ifndef MACRO

#define MACRO( opcode, layout, attr)

#endif

#ifndef MACRO\_WMS

#define MACRO\_WMS(opcode, layout, attr)

#endif

#ifndef MACRO\_EXTEND

#define MACRO\_EXTEND(opcode, layout, attr)

#endif

#ifndef MACRO\_EXTEND\_WMS

#define MACRO\_EXTEND\_WMS(opcode, layout, attr)

#endif

#ifndef MACRO\_BACKEND\_ONLY

#define MACRO\_BACKEND\_ONLY(opcode, layout, attr)

#endif

#define MACRO\_WMS\_PROFILED( opcode, layout, attr) \

MACRO\_WMS(opcode, layout, OpHasProfiled|attr) \

MACRO\_WMS(Profiled##opcode, Profiled##layout, OpByteCodeOnly|OpProfiled|attr) \

#define MACRO\_WMS\_PROFILED2(opcode, layout, attr) \

MACRO\_WMS(opcode, layout, OpHasProfiled|attr) \

MACRO\_WMS(Profiled##opcode, Profiled2##layout, OpByteCodeOnly|OpProfiled|attr) \

#define MACRO\_WMS\_PROFILED\_INDEX(opcode, layout, attr) \

MACRO\_WMS(opcode, layout, OpHasProfiled|OpHasProfiledWithICIndex|attr) \

MACRO\_WMS(Profiled##opcode, Profiled##layout, OpByteCodeOnly|OpProfiled|attr) \

MACRO\_WMS(Profiled##opcode##WithICIndex, Profiled##layout##WithICIndex, OpByteCodeOnly|OpProfiledWithICIndex|attr) \

#define MACRO\_WMS\_PROFILED\_OP( opcode, layout, attr) \

MACRO\_WMS(opcode, layout, OpHasProfiled|attr) \

MACRO\_WMS(Profiled##opcode, layout, OpByteCodeOnly|OpProfiled|attr) \

#define MACRO\_EXTEND\_WMS\_AND\_PROFILED\_OP(opcode, layout, attr) \

MACRO\_EXTEND\_WMS(opcode, layout, OpHasProfiled | attr) \

MACRO\_EXTEND\_WMS(Profiled##opcode, layout, OpByteCodeOnly | OpProfiled | attr) \

#define MACRO\_PROFILED(opcode, layout, attr) \

MACRO(opcode, layout, OpHasProfiled|attr) \

MACRO(Profiled##opcode, Profiled##layout, OpByteCodeOnly|OpProfiled|attr) \

#define MACRO\_EXTEND\_WMS\_AND\_PROFILED(opcode, layout, attr) \

MACRO\_EXTEND\_WMS(opcode, layout, OpHasProfiled|attr) \

MACRO\_EXTEND\_WMS(Profiled##opcode, Profiled##layout, OpByteCodeOnly|OpProfiled|attr) \

MACRO( EndOfBlock, Empty, OpByteCodeOnly) // End-of-buffer

MACRO( ExtendedOpcodePrefix,Empty, OpByteCodeOnly)

MACRO( MediumLayoutPrefix, Empty, OpByteCodeOnly)

MACRO( ExtendedMediumLayoutPrefix,Empty, OpByteCodeOnly)

MACRO( LargeLayoutPrefix, Empty, OpByteCodeOnly)

MACRO( ExtendedLargeLayoutPrefix,Empty, OpByteCodeOnly)

MACRO( Nop, Empty, None) // No operation (Default value = 0)

MACRO( StartCall, StartCall, OpSideEffect)

MACRO\_BACKEND\_ONLY( LoweredStartCall, StartCall, OpSideEffect) // StartCall instruction after it's been lowered

MACRO\_BACKEND\_ONLY( StartCallAsmJsI, StartCall, OpSideEffect) // StartCall instruction for asm.js internal calls

MACRO\_BACKEND\_ONLY( StartCallAsmJsE, StartCall, OpSideEffect) // StartCall instruction for calls from asm.js to javascript

MACRO( Break, Empty, OpSideEffect) // Break into debugger

MACRO\_EXTEND( InvalidOpCode, Empty, None) // Inserted in a dead call sequence, should not be present after GlobOpt

// Control flow

#ifdef BYTECODE\_BRANCH\_ISLAND

MACRO\_EXTEND( BrLong, BrLong, OpByteCodeOnly|OpSideEffect|OpNoFallThrough)

#endif

MACRO( Br , Br, OpSideEffect|OpNoFallThrough) // Unconditional branch

MACRO\_BACKEND\_ONLY( MultiBr, Empty, OpSideEffect|OpNoFallThrough) // Unconditional multi-target branch

MACRO\_BACKEND\_ONLY( BrHasSideEffects, BrS, OpSideEffect|OpTempNumberSources) // Branch if there are any of the side effects in val

MACRO( BrNotHasSideEffects,BrS, OpSideEffect|OpTempNumberSources) // Branch if there are none of the side effects in val

MACRO\_BACKEND\_ONLY( BrFncEqApply , BrReg1, OpSideEffect|OpTempNumberSources) // Branch if R1 is func whose native code ptr == Apply

MACRO\_WMS( BrFncNeqApply , BrReg1, OpSideEffect|OpTempNumberSources) // Branch if R1 is func whose native code ptr != Apply

MACRO\_WMS( BrEq\_A, BrReg2, OpSideEffect|OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources) // Branch if '==' (general equals)

MACRO\_WMS( BrNeq\_A, BrReg2, OpSideEffect|OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources) // Branch if '!=' (not general equals)

MACRO\_WMS( BrFalse\_A, BrReg1, OpSideEffect|OpTempNumberSources|OpTempObjectSources) // Branch if 'false' (NOTE: do not call ToPrimitive, no implicit call)

MACRO\_WMS( BrTrue\_A, BrReg1, OpSideEffect|OpTempNumberSources|OpTempObjectSources) // Branch if 'true' (NOTE: do not call ToPrimitive, no implicit call)

MACRO\_WMS( BrGe\_A, BrReg2, OpSideEffect|OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources|OpPostOpDbgBailOut) // Branch if '>='

MACRO\_BACKEND\_ONLY( BrUnGe\_A, BrReg2, OpSideEffect|OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources|OpPostOpDbgBailOut) // Branch if '>='

MACRO\_WMS( BrGt\_A, BrReg2, OpSideEffect|OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources|OpPostOpDbgBailOut) // Branch if '>'

MACRO\_BACKEND\_ONLY( BrUnGt\_A, BrReg2, OpSideEffect|OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources|OpPostOpDbgBailOut) // Branch if '>'

MACRO\_WMS( BrLt\_A, BrReg2, OpSideEffect|OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources|OpPostOpDbgBailOut) // Branch if '<'

MACRO\_BACKEND\_ONLY( BrUnLt\_A, BrReg2, OpSideEffect|OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources|OpPostOpDbgBailOut) // Branch if '<'

MACRO\_WMS( BrLe\_A, BrReg2, OpSideEffect|OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources|OpPostOpDbgBailOut) // Branch if '<='

MACRO\_BACKEND\_ONLY( BrUnLe\_A, BrReg2, OpSideEffect|OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources|OpPostOpDbgBailOut) // Branch if '<='

MACRO\_WMS( BrSrEq\_A, BrReg2, OpSideEffect|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources) // Branch if '===' (strict equals) (NOTE: May have DOM implicit calls)

MACRO\_WMS( BrSrNeq\_A, BrReg2, OpSideEffect|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources) // Branch if '!==' (not strict equals) (NOTE: May have DOM implicit calls)

MACRO\_EXTEND( BrOnHasProperty, BrProperty, OpSideEffect|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources) // Branch if object has the given property (NOTE: May have DOM implicit calls)

MACRO( BrOnNoProperty, BrProperty, OpSideEffect|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources) // Branch if object does not have the given property (NOTE: May have DOM implicit calls)

MACRO( BrOnNoLocalProperty,BrLocalProperty,OpSideEffect|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources) // Branch if object does not have the given property (NOTE: May have DOM implicit calls)

MACRO( BrOnNoEnvProperty, BrEnvProperty, OpSideEffect|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources) // Branch if object does not have the given property (NOTE: May have DOM implicit calls)

MACRO\_WMS( BrOnObject\_A, BrReg1, OpSideEffect|OpTempNumberSources|OpTempObjectSources) // Branch if typeId is not primitive type (i.e. > TypeIds\_LastJavascriptPrimitiveType)

MACRO\_WMS( BrNotNull\_A, BrReg1, OpSideEffect|OpTempNumberSources|OpTempObjectSources) // Branch if not NULL

MACRO\_EXTEND\_WMS( BrNotUndecl\_A, BrReg1, OpSideEffect|OpTempNumberSources|OpTempObjectSources) // Branch if source reg is NEQ to Undecl

MACRO\_BACKEND\_ONLY( BrNotEq\_A, Empty, OpSideEffect|OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources) // Branch if !'>='

MACRO\_BACKEND\_ONLY( BrNotNeq\_A, Empty, OpSideEffect|OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources) // Branch if !'>='

MACRO\_BACKEND\_ONLY( BrSrNotEq\_A, Empty, OpSideEffect|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources) // Branch if !'>='

MACRO\_BACKEND\_ONLY( BrSrNotNeq\_A, Empty, OpSideEffect|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources) // Branch if !'>='

MACRO\_BACKEND\_ONLY( BrNotGe\_A, Empty, OpSideEffect|OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources|OpPostOpDbgBailOut) // Branch if !'>='

MACRO\_BACKEND\_ONLY( BrNotGt\_A, Empty, OpSideEffect|OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources|OpPostOpDbgBailOut) // Branch if !'>'

MACRO\_BACKEND\_ONLY( BrNotLt\_A, Empty, OpSideEffect|OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources|OpPostOpDbgBailOut) // Branch if !'<'

MACRO\_BACKEND\_ONLY( BrNotLe\_A, Empty, OpSideEffect|OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources|OpPostOpDbgBailOut) // Branch if !'<='

MACRO\_BACKEND\_ONLY( BrNotAddr\_A, BrReg2, OpSideEffect|OpTempNumberSources|OpTempObjectSources) // Branch if != Address

MACRO\_BACKEND\_ONLY( BrAddr\_A, BrReg2, OpSideEffect|OpTempNumberSources|OpTempObjectSources) // Branch if == Address

MACRO\_BACKEND\_ONLY( BrOnException, Br, OpSideEffect)

MACRO\_BACKEND\_ONLY( BrOnNoException, Br, OpSideEffect) // Mark it OpNoFallThrough?

MACRO\_BACKEND\_ONLY( DeadBrEqual, Reg3, OpSideEffect|OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources) // Dead BrEqual, but still need to reference srcs

MACRO\_BACKEND\_ONLY( DeadBrSrEqual, Reg3, OpSideEffect|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources) // Dead BrSrEqual, but still need to reference srcs

MACRO\_BACKEND\_ONLY( DeadBrRelational, Reg3, OpSideEffect|OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources) // Dead BrRelational, but still need to reference srcs

MACRO\_BACKEND\_ONLY( DeadBrOnHasProperty,Reg2, OpSideEffect|OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources) // Dead BrOnHasProperty, but still need to reference srcs

MACRO\_BACKEND\_ONLY( DeletedNonHelperBranch,Empty, OpSideEffect) // Deleted branch to non-helper. Used to avoid opHleper asserts.

MACRO\_BACKEND\_ONLY( BrEq\_I4, Empty, OpSideEffect|OpTempNumberSources) // int32 Branch if '==' (general equals)

MACRO\_BACKEND\_ONLY( BrTrue\_I4, Empty, OpSideEffect|OpTempNumberSources) // int32 Branch if 'true'

MACRO\_BACKEND\_ONLY( BrFalse\_I4, Empty, OpSideEffect|OpTempNumberSources) // int32 Branch if 'false'

MACRO\_BACKEND\_ONLY( BrGe\_I4, Empty, OpSideEffect|OpTempNumberSources) // int32 Branch if '>='

MACRO\_BACKEND\_ONLY( BrUnGe\_I4, Empty, OpSideEffect|OpTempNumberSources) // uint32 Branch if '>='

MACRO\_BACKEND\_ONLY( BrGt\_I4, Empty, OpSideEffect|OpTempNumberSources) // int32 Branch if '>'

MACRO\_BACKEND\_ONLY( BrUnGt\_I4, Empty, OpSideEffect|OpTempNumberSources) // uint32 Branch if '>'

MACRO\_BACKEND\_ONLY( BrLt\_I4, Empty, OpSideEffect|OpTempNumberSources) // int32 Branch if '<'

MACRO\_BACKEND\_ONLY( BrUnLt\_I4, Empty, OpSideEffect|OpTempNumberSources) // uint32 Branch if '<'

MACRO\_BACKEND\_ONLY( BrLe\_I4, Empty, OpSideEffect|OpTempNumberSources) // int32 Branch if '<='

MACRO\_BACKEND\_ONLY( BrUnLe\_I4, Empty, OpSideEffect|OpTempNumberSources) // uint32 Branch if '<='

MACRO\_BACKEND\_ONLY( BrNeq\_I4, Empty, OpSideEffect|OpTempNumberSources) // int32 Branch if '!=' (not general equals)

// For Switch Case optimization

MACRO( EndSwitch, Br, OpSideEffect) // Unconditional branch - This is equivalent to Br OpCode

MACRO\_WMS( Case, BrReg2, OpSideEffect|OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources) // Equivalent to Branch if '===' (strict equals) (NOTE: May have DOM implicit calls)

MACRO\_WMS\_PROFILED( BeginSwitch, Reg2, OpTempNumberTransfer|OpTempObjectTransfer|OpNonIntTransfer) // Switch statement - identifies the start of a switch statement

// performs the function of Ld\_A and does profiling

// All the Call\* function need to be together so that ProfileCall\* can be calculated

MACRO\_BACKEND\_ONLY( Call, Reg1, OpSideEffect|OpUseAllFields|OpCallInstr) // R0 <- Call (direct) registered function

MACRO\_BACKEND\_ONLY( AsmJsCallI, Reg1, OpSideEffect|OpUseAllFields|OpCallInstr) // call from asm.js to asm.js

MACRO\_BACKEND\_ONLY( AsmJsCallE, Reg1, OpSideEffect|OpUseAllFields|OpCallInstr) // call from asm.js to javascript

// CallI through CallIExtendedFlags need to stay in this order since all the ProfiledCall\* opcodes are calculated based on this order

MACRO\_WMS( CallI, CallI, OpSideEffect|OpUseAllFields|OpCallInstr|OpInlineCallInstr) // Return <- Call (indirect) Function(ArgCount)

MACRO\_WMS( CallIFlags, CallIFlags, OpSideEffect|OpUseAllFields|OpCallInstr|OpInlineCallInstr) // Return <- Call (indirect) Function(ArgCount)

MACRO\_WMS( CallIExtended, CallIExtended, OpSideEffect|OpUseAllFields|OpCallInstr)

MACRO\_WMS( CallIExtendedFlags, CallIExtendedFlags, OpSideEffect|OpUseAllFields|OpCallInstr)

MACRO\_BACKEND\_ONLY( CallIPut, CallIFlags, OpSideEffect|OpUseAllFields|OpCallInstr) // Call (indirect) Function(ArgCount) to put value

MACRO\_BACKEND\_ONLY( CallINew, CallIFlags, OpSideEffect|OpUseAllFields|OpCallInstr)

MACRO\_BACKEND\_ONLY( CallINewTargetNew, CallIFlags, OpSideEffect|OpUseAllFields|OpCallInstr)

MACRO\_BACKEND\_ONLY( CallIExtendedNew, CallIExtendedFlags, OpSideEffect|OpUseAllFields|OpCallInstr)

MACRO\_BACKEND\_ONLY( CallIEval, CallIExtendedFlags, OpSideEffect|OpUseAllFields|OpCallInstr)

MACRO\_BACKEND\_ONLY( CallIExtendedNewTargetNew, CallIExtendedFlags, OpSideEffect|OpUseAllFields|OpCallInstr)

MACRO\_BACKEND\_ONLY( CallIDynamic, CallI, OpSideEffect|OpUseAllFields|OpCallInstr)

MACRO\_BACKEND\_ONLY( CallIDynamicSpread, CallI, OpSideEffect|OpUseAllFields|OpCallInstr)

MACRO\_BACKEND\_ONLY( CallDirect, Empty, OpTempNumberSources|OpCallInstr|OpSideEffect|OpHasImplicitCall|OpTempObjectProducing) // For direct calls to helper (used in inlining built-ins)

MACRO\_BACKEND\_ONLY( InlineThrow, Reg1, OpSideEffect|OpPostOpDbgBailOut|OpDeadFallThrough) // Throw exception

MACRO\_BACKEND\_ONLY( EHThrow, Reg1, OpSideEffect|OpPostOpDbgBailOut|OpDeadFallThrough) // Throw exception

MACRO\_WMS( Throw, Reg1, OpSideEffect|OpNoFallThrough|OpPostOpDbgBailOut) // Throw exception

MACRO( Ret, Empty, OpSideEffect|OpUseAllFields|OpNoFallThrough) // Return from function

MACRO\_WMS( Yield, Reg2, OpSideEffect|OpUseAllFields) // Yield from generator function

MACRO\_WMS( ResumeYield, Reg2, OpSideEffect)

MACRO\_WMS( ResumeYieldStar, Reg3, OpSideEffect)

MACRO\_EXTEND\_WMS( AsyncSpawn, Reg3, OpSideEffect|OpUseAllFields)

// Unary operations

MACRO\_WMS( Incr\_A, Reg2, OpTempNumberProducing|OpCallsValueOf|OpDoNotTransfer|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut|OpProducesNumber) // Increment

MACRO\_WMS( Decr\_A, Reg2, OpTempNumberProducing|OpCallsValueOf|OpDoNotTransfer|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut|OpProducesNumber) // Decrement

MACRO\_WMS( Neg\_A, Reg2, OpTempNumberProducing|OpCallsValueOf|OpDoNotTransfer|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut|OpProducesNumber) // Arithmetic '-' (negate)

MACRO\_WMS( Not\_A, Reg2, OpTempNumberProducing|OpCallsValueOf|OpDoNotTransfer|OpIsInt32|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut|OpProducesNumber) // Boolean '!' (not)

// Integer Unary operations

MACRO\_BACKEND\_ONLY( Neg\_I4, Empty, OpDoNotTransfer|OpTempNumberSources|OpCanCSE) // int32 Arithmetic '-' (negate)

MACRO\_BACKEND\_ONLY( Not\_I4, Empty, OpDoNotTransfer|OpTempNumberSources|OpCanCSE) // int32 '!' (not)

// NOTE:

// Typeof

// Only have external call for HostDispatch, but only QI, and shouldn't have direct side effect,

// so we can dead store it but still need to check for implicit call.

// It also doesn't call valueOf/toString

// TypeofElem

// May throw exception if x (in x.y or x[y]) is null or undefined, so it can't be dead store (OpSideEffect)

// GetProperty may call getter, Typeof may have external call for HostDispatch

// It also doesn't call valueOf/toString

// REVIEW: TypeofRootFld will be off the root object which can't be null? So we could remove OpSideEffect?

MACRO\_WMS( Typeof, Reg2, OpTempNumberProducing|OpDoNotTransfer|OpHasImplicitCall|OpCanCSE) // typeof x

MACRO\_WMS( TypeofElem, ElementI, OpTempNumberProducing|OpSideEffect|OpDoNotTransfer|OpHasImplicitCall|OpPostOpDbgBailOut) // typeof x[y]

MACRO\_WMS( ApplyArgs, Reg5, OpByteCodeOnly|OpSideEffect|OpUseAllFields) // apply function to this function's arguments and this pointer

// Binary operations

MACRO\_WMS( Add\_A, Reg3, OpTempNumberProducing|OpCallsValueOf|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut) // Arithmetic '+'

MACRO\_WMS\_PROFILED( Div\_A, Reg3, OpTempNumberProducing|OpCallsValueOf|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut|OpProducesNumber) // Arithmetic '/'

MACRO\_WMS( Mul\_A, Reg3, OpTempNumberProducing|OpCallsValueOf|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut|OpProducesNumber) // Arithmetic '\*'

MACRO\_WMS\_PROFILED( Rem\_A, Reg3, OpTempNumberProducing|OpCallsValueOf|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut|OpProducesNumber) // Arithmetic '%'

MACRO\_WMS( Sub\_A, Reg3, OpTempNumberProducing|OpCallsValueOf|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut|OpProducesNumber) // Arithmetic '-' (subtract)

MACRO\_WMS( Expo\_A, Reg3, OpTempNumberProducing|OpCallsValueOf|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut|OpProducesNumber) // Arithmetic '\*\*' (exponentiation)

MACRO\_WMS( And\_A, Reg3, OpTempNumberProducing|OpCallsValueOf|OpIsInt32|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut|OpProducesNumber) // Bitwise '&'

MACRO\_WMS( Or\_A, Reg3, OpTempNumberProducing|OpCallsValueOf|OpIsInt32|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut|OpProducesNumber) // Bitwise '|'

MACRO\_WMS( Xor\_A, Reg3, OpTempNumberProducing|OpCallsValueOf|OpIsInt32|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut|OpProducesNumber) // Bitwise '^'

MACRO\_WMS( Shl\_A, Reg3, OpTempNumberProducing|OpCallsValueOf|OpIsInt32|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut|OpProducesNumber) // Shift '<<' (signed, truncate)

MACRO\_WMS( Shr\_A, Reg3, OpTempNumberProducing|OpCallsValueOf|OpIsInt32|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpProducesNumber) // Shift '>>' (signed, truncate)

MACRO\_WMS( ShrU\_A, Reg3, OpTempNumberProducing|OpCallsValueOf|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut|OpProducesNumber) // Shift '>>>'(unsigned, truncate)

// Integer Binary operations used only by backend, hence no need for one byte variant

MACRO\_BACKEND\_ONLY( Add\_I4, Empty, OpTempNumberSources|OpCanCSE) // int32 Arithmetic '+'

MACRO\_BACKEND\_ONLY( Sub\_I4, Empty, OpTempNumberSources|OpCanCSE) // int32 Arithmetic '-' (subtract)

MACRO\_BACKEND\_ONLY( Mul\_I4, Empty, OpTempNumberSources|OpCanCSE) // int32 Arithmetic '\*'

MACRO\_BACKEND\_ONLY( Div\_I4, Empty, OpTempNumberSources|OpCanCSE) // int32 Arithmetic '/'

MACRO\_BACKEND\_ONLY( Rem\_I4, Empty, OpTempNumberSources|OpCanCSE) // int32 Arithmetic '%'

MACRO\_BACKEND\_ONLY( And\_I4, Empty, OpTempNumberSources|OpCanCSE) // int32 Bitwise '&'

MACRO\_BACKEND\_ONLY( Or\_I4, Empty, OpTempNumberSources|OpCanCSE) // int32 Bitwise '|'

MACRO\_BACKEND\_ONLY( Xor\_I4, Empty, OpTempNumberSources|OpCanCSE) // int32 Bitwise '^'

MACRO\_BACKEND\_ONLY( Shl\_I4, Empty, OpTempNumberSources|OpCanCSE) // int32 Shift '<<' (signed, truncate)

MACRO\_BACKEND\_ONLY( Shr\_I4, Empty, OpTempNumberSources|OpCanCSE) // int32 Shift '>>' (signed, truncate)

MACRO\_BACKEND\_ONLY( ShrU\_I4, Empty, OpTempNumberSources|OpCanCSE) // int32 Shift '>>>'(unsigned, truncate)

MACRO\_BACKEND\_ONLY( Add\_Ptr, Empty, OpTempNumberSources|OpCanCSE) // ptr Arithmetic '+'

// Comparison

MACRO\_WMS( CmEq\_A, Reg3, OpCallsValueOf|OpTempNumberSources|OpTempObjectSources|OpCanCSE) // Compare if '==' (general equals)

MACRO\_WMS( CmGe\_A, Reg3, OpCallsValueOf|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut) // Compare if '>='

MACRO\_BACKEND\_ONLY( CmUnGe\_A, Reg3, OpCallsValueOf|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut) // Compare if '>='

MACRO\_WMS( CmGt\_A, Reg3, OpCallsValueOf|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut) // Compare if '>'

MACRO\_BACKEND\_ONLY( CmUnGt\_A, Reg3, OpCallsValueOf|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut) // Compare if '>'

MACRO\_WMS( CmLt\_A, Reg3, OpCallsValueOf|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut) // Compare if '<'

MACRO\_BACKEND\_ONLY( CmUnLt\_A, Reg3, OpCallsValueOf|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut) // Compare if '<'

MACRO\_WMS( CmLe\_A, Reg3, OpCallsValueOf|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut) // Compare if '<='

MACRO\_BACKEND\_ONLY( CmUnLe\_A, Reg3, OpCallsValueOf|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut) // Compare if '<='

MACRO\_WMS( CmNeq\_A, Reg3, OpCallsValueOf|OpTempNumberSources|OpTempObjectSources|OpCanCSE) // Compare if '!=' (not general equals)

MACRO\_WMS( CmSrEq\_A, Reg3, OpCallsValueOf|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpTempObjectSources) // Compare if '===' (strict equals) (NOTE: May have DOM implicit calls)

MACRO\_WMS( CmSrNeq\_A, Reg3, OpCallsValueOf|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpTempObjectSources) // Compare if '!==' (not strict equals) (NOTE: May have DOM implicit calls)

MACRO\_BACKEND\_ONLY( CmEq\_I4, Reg3, OpTempNumberSources|OpCanCSE) // Unsigned I4 Compare if '=='

MACRO\_BACKEND\_ONLY( CmNeq\_I4, Reg3, OpTempNumberSources|OpCanCSE) // Unsigned I4 Compare if '!='

MACRO\_BACKEND\_ONLY( CmLt\_I4, Reg3, OpTempNumberSources|OpCanCSE) // Unsigned I4 Compare if '<'

MACRO\_BACKEND\_ONLY( CmLe\_I4, Reg3, OpTempNumberSources|OpCanCSE) // Unsigned I4 Compare if '<='

MACRO\_BACKEND\_ONLY( CmGt\_I4, Reg3, OpTempNumberSources|OpCanCSE) // Unsigned I4 Compare if '>'

MACRO\_BACKEND\_ONLY( CmGe\_I4, Reg3, OpTempNumberSources|OpCanCSE) // Unsigned I4 Compare if '>='

MACRO\_BACKEND\_ONLY( CmUnLt\_I4, Reg3, OpTempNumberSources|OpCanCSE) // Unsigned I4 Compare if '<'

MACRO\_BACKEND\_ONLY( CmUnLe\_I4, Reg3, OpTempNumberSources|OpCanCSE) // Unsigned I4 Compare if '<='

MACRO\_BACKEND\_ONLY( CmUnGt\_I4, Reg3, OpTempNumberSources|OpCanCSE) // Unsigned I4 Compare if '>'

MACRO\_BACKEND\_ONLY( CmUnGe\_I4, Reg3, OpTempNumberSources|OpCanCSE) // Unsigned I4 Compare if '>='

// Conversions

MACRO\_WMS( Conv\_Num, Reg2, OpSideEffect|OpTempNumberProducing|OpTempNumberTransfer|OpTempObjectSources|OpCallsValueOf|OpProducesNumber) // Convert to Number. [[ToNumber()]]

// Operation ToString(str)

MACRO\_EXTEND\_WMS( Conv\_Str, Reg2, OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut)

// Conv\_Obj:

// OpSideEffect - May throw exception on null/undefined.

// Do not call valueOf/toString no implicit call

MACRO\_WMS( Conv\_Obj, Reg2, OpSideEffect|OpPostOpDbgBailOut|OpTempObjectTransfer) // Convert to Object

MACRO\_EXTEND\_WMS( NewWithObject, Reg2, OpSideEffect | OpPostOpDbgBailOut) // Wrap in a with Object

MACRO\_BACKEND\_ONLY( ToVar, Reg2, OpTempNumberProducing|OpTempNumberSources|OpCanCSE) // Load from int32/float64 to Var(reg)

// Load from Var(reg) to int32/float64, NOTE: always bail if it is not primitive. so no implicit call, but still mark with CallsValueOf so it won't get automatically dead stored

// TODO: Consider changing the code so we don't have mark this as CallsValueOf

MACRO\_BACKEND\_ONLY( FromVar, Reg2, OpTempNumberSources|OpTempObjectSources|OpCanCSE)

MACRO\_BACKEND\_ONLY( Conv\_Prim, Reg2, OpTempNumberProducing|OpTempNumberSources|OpCanCSE|OpPostOpDbgBailOut) // Convert between primitives (int32/float64)

MACRO\_BACKEND\_ONLY( Conv\_Bool, Reg2, OpTempNumberSources|OpCanCSE) // Convert from i4 to bool

// Register

MACRO\_EXTEND\_WMS( UnwrapWithObj, Reg2, OpSideEffect) // Copy Var register with unwrapped object

MACRO\_EXTEND\_WMS( SetComputedNameVar, Reg2, OpSideEffect)

MACRO\_WMS( Ld\_A, Reg2, OpTempNumberTransfer|OpTempObjectTransfer|OpNonIntTransfer|OpCanCSE) // Copy Var register

MACRO\_WMS( LdLocalObj, Reg1, OpCanCSE) // Load non-stack frame object

MACRO\_WMS( LdInnerScope, Reg1Unsigned1, OpCanCSE) // Load non-stack inner scope

MACRO\_WMS( LdC\_A\_Null, Reg1, OpByteCodeOnly|OpCanCSE) // Load from 'null' as Var

MACRO\_BACKEND\_ONLY( Ld\_I4, Empty, OpCanCSE) // Copy I4 register

MACRO\_BACKEND\_ONLY( LdC\_A\_I4, Empty, OpCanCSE) // Load from 'int32' as Var(C)

MACRO\_BACKEND\_ONLY( LdC\_A\_R8, Empty, OpCanCSE) // Load from 'double' constant

MACRO\_BACKEND\_ONLY( LdC\_F8\_R8, Empty, OpCanCSE) // Load 'double' constant

MACRO\_BACKEND\_ONLY( LdIndir, Empty, OpTempNumberSources|OpTempObjectSources|OpCanCSE)

MACRO\_WMS( ChkUndecl, Reg1, OpSideEffect)

MACRO\_WMS( EnsureNoRootFld, ElementRootU, OpSideEffect|OpHasImplicitCall)

MACRO\_WMS( EnsureNoRootRedeclFld, ElementRootU, OpSideEffect|OpHasImplicitCall)

MACRO\_WMS( ScopedEnsureNoRedeclFld, ElementScopedC, OpSideEffect|OpHasImplicitCall)

MACRO\_WMS( InitUndecl, Reg1, OpCanCSE)

//MACRO\_WMS( InitUndeclSlot, Reg1Unsigned1, None)

// TODO: Change InitUndeclLetFld and InitUndeclConstFld to ElementU layouts since they do not use their inline cache

MACRO\_WMS( InitUndeclLetFld, ElementPIndexed,OpByteCodeOnly|OpSideEffect|OpHasImplicitCall)

MACRO\_EXTEND\_WMS( InitUndeclLocalLetFld, ElementP, OpByteCodeOnly|OpSideEffect|OpHasImplicitCall)

MACRO\_WMS( InitUndeclRootLetFld, ElementRootU, OpSideEffect|OpHasImplicitCall)

MACRO\_WMS( InitUndeclConstFld, ElementPIndexed,OpByteCodeOnly|OpSideEffect|OpHasImplicitCall)

MACRO\_EXTEND\_WMS( InitUndeclLocalConstFld, ElementP, OpByteCodeOnly|OpSideEffect|OpHasImplicitCall)

MACRO\_WMS( InitUndeclRootConstFld, ElementRootU, OpSideEffect|OpHasImplicitCall)

MACRO\_EXTEND\_WMS( InitUndeclConsoleLetFld, ElementScopedU, OpSideEffect|OpHasImplicitCall)

MACRO\_EXTEND\_WMS( InitUndeclConsoleConstFld, ElementScopedU, OpSideEffect|OpHasImplicitCall)

MACRO\_WMS( InitConst, Reg2, OpTempNumberTransfer|OpTempObjectTransfer|OpNonIntTransfer|OpCanCSE) // Create and initialize 'const' as property of global object

MACRO\_WMS( InitConstSlot, ElementSlot, None)

MACRO\_WMS( InitLetFld, ElementCP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut) // Declare a property with an initial value

MACRO\_WMS( InitRootLetFld, ElementRootCP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut) // Declare a property with an initial value

MACRO\_WMS( InitConstFld, ElementCP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut) // Declare a property with an initial value

MACRO\_WMS( InitRootConstFld, ElementRootCP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut) // Declare a property with an initial value

MACRO\_WMS( InitClassMember, ElementCP, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Class member

MACRO\_EXTEND\_WMS( InitClassMemberComputedName,ElementI, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Class member with computed property name

MACRO\_EXTEND\_WMS( InitClassMemberSet, ElementC, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Class member in set syntax

MACRO\_EXTEND\_WMS( InitClassMemberGet, ElementC, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Class member in get syntax

MACRO\_EXTEND\_WMS( InitClassMemberSetComputedName,ElementI, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Class member in set syntax with computed property name

MACRO\_EXTEND\_WMS( InitClassMemberGetComputedName,ElementI, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Class member in get syntax with computed property name

MACRO\_EXTEND\_WMS( BrOnClassConstructor, BrReg1, None) // Branch if argument is a class constructor

MACRO\_BACKEND\_ONLY( ArgIn\_A, Empty, OpSideEffect) // Copy from "in slot" to "local slot", unchecked

MACRO\_WMS( ArgIn0, Reg1, OpByteCodeOnly) // Copy from "in slot" to "local slot", unchecked

MACRO\_BACKEND\_ONLY( ArgIn\_Rest, Empty, OpSideEffect)

MACRO\_WMS\_PROFILED( ArgOut\_A, Arg, OpSideEffect) // Copy from "local slot" to "out slot"

MACRO\_BACKEND\_ONLY( ArgOut\_A\_Inline, Empty, OpSideEffect|OpTempNumberSources|OpTempObjectSources) // Copy from "local slot" to "out slot"

MACRO\_BACKEND\_ONLY( ArgOut\_A\_InlineBuiltIn, Empty, OpSideEffect|OpTempNumberTransfer|OpTempObjectTransfer) // Used as virtual ArgOut for inline built-ins (Math.sin, etc). Removed in lowerer.

MACRO\_BACKEND\_ONLY( ArgOut\_A\_InlineSpecialized, Empty, OpSideEffect) // Used as virtual ArgOut to keep function object around for inlining built-ins by direct call to helpers.

MACRO\_WMS( ArgOut\_ANonVar, Arg, OpByteCodeOnly|OpSideEffect) // Copy from "local slot" to "out slot"

MACRO\_WMS( ArgOut\_Env, ArgNoSrc, OpByteCodeOnly|OpSideEffect) // Copy from "local slot" to "out slot"

MACRO\_BACKEND\_ONLY( ArgOut\_A\_Dynamic, Empty, OpSideEffect) // Copy from "local slot" to "out slot"

MACRO\_BACKEND\_ONLY( ArgOut\_A\_FromStackArgs, Empty, OpSideEffect) // Copy from "local slot" to "out slot"

MACRO\_BACKEND\_ONLY( ArgOut\_A\_FixupForStackArgs, Empty, OpSideEffect)

MACRO\_BACKEND\_ONLY( ArgOut\_A\_SpreadArg, Empty, OpSideEffect)

MACRO\_BACKEND\_ONLY( ArgOutAsmJsI\_A, Empty, OpSideEffect)

MACRO\_BACKEND\_ONLY( ArgOutAsmJsE\_A, Empty, OpSideEffect)

MACRO\_WMS( Delete\_A, Reg2, OpSideEffect|OpPostOpDbgBailOut) // Delete Var

// Object operations

MACRO\_WMS\_PROFILED\_OP( LdFld, ElementCP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut|OpCanLoadFixedFields) // Load from ScriptObject instance's direct field

MACRO\_WMS\_PROFILED\_OP( LdLocalFld, ElementP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut|OpCanLoadFixedFields) // Load from ScriptObject instance's direct field

MACRO\_WMS( LdEnvObj, ElementSlotI1, OpTempNumberSources)

MACRO\_EXTEND\_WMS\_AND\_PROFILED\_OP(LdSuperFld, ElementC2, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut|OpCanLoadFixedFields) // Load from ScriptObject super instance's direct field

MACRO\_WMS\_PROFILED\_OP( LdFldForTypeOf, ElementCP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut|OpCanLoadFixedFields)

MACRO\_EXTEND\_WMS\_AND\_PROFILED\_OP(LdRootFldForTypeOf, ElementRootCP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut|OpCanLoadFixedFields)

MACRO\_WMS\_PROFILED\_OP( LdFldForCallApplyTarget, ElementCP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut)

MACRO\_WMS\_PROFILED\_OP(LdRootFld, ElementRootCP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut|OpCanLoadFixedFields) // Load from ScriptObject instance's direct field (access to let/const on root object)

MACRO\_WMS\_PROFILED\_OP(LdMethodFld, ElementCP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut|OpCanLoadFixedFields) // Load call target from ScriptObject instance's direct field

MACRO\_EXTEND\_WMS\_AND\_PROFILED\_OP(LdLocalMethodFld, ElementP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut|OpCanLoadFixedFields) // Load call target from ScriptObject instance's direct field

MACRO\_BACKEND\_ONLY( LdMethodFldPolyInlineMiss, ElementCP, OpSideEffect|OpHasImplicitCall|OpDoNotTransfer|OpPostOpDbgBailOut) // Load call target from ScriptObject instance's direct field, when the call target is neither of

// the ones we inlined using fixed methods, at a polymorphic call site,

// but don't allow it to participate in any obj type spec optimizations,

// as it will always result in a helper call.

MACRO\_WMS\_PROFILED\_OP( LdRootMethodFld, ElementRootCP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut) // Load call target from ScriptObject instance's direct field (access to let/const on root object)

MACRO\_WMS\_PROFILED\_OP( StFld, ElementCP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut) // Store into ScriptObject instance's direct field

MACRO\_EXTEND\_WMS\_AND\_PROFILED\_OP(StSuperFld, ElementC2, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut) // Store into ScriptObject super instance's direct field

MACRO\_WMS\_PROFILED\_OP( StRootFld, ElementRootCP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut) // Store into ScriptObject instance's direct field (access to let/const on root object)

MACRO\_WMS\_PROFILED\_OP( StLocalFld, ElementP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut) // Store into local activation object

MACRO\_WMS\_PROFILED\_OP( StFldStrict, ElementCP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut) // Store into ScriptObject instance's direct field (strict mode, a.x = ...)

MACRO\_WMS\_PROFILED\_OP( StRootFldStrict, ElementRootCP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut) // Store into ScriptObject instance's direct field (strict mode, x = ..., access to let/const on root object)

MACRO\_WMS\_PROFILED\_OP( InitFld, ElementCP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut) // Declare a property with an initial value

MACRO\_WMS\_PROFILED\_OP( InitLocalFld, ElementP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut) // Declare a property with an initial value

MACRO\_EXTEND\_WMS( InitLocalLetFld, ElementP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut) // Declare a property with an initial value

MACRO\_EXTEND\_WMS( InitInnerFld, ElementPIndexed,OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut) // Declare a property with an initial value

MACRO\_EXTEND\_WMS( InitInnerLetFld, ElementPIndexed,OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut) // Declare a property with an initial value

MACRO\_WMS\_PROFILED\_OP( InitRootFld, ElementRootCP, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut) // Declare a property with an initial value

MACRO\_BACKEND\_ONLY( LdMethodFromFlags, ElementCP, OpFastFldInstr|OpCanCSE)

MACRO\_WMS( DeleteFld, ElementC, OpSideEffect|OpHasImplicitCall|OpDoNotTransfer|OpPostOpDbgBailOut) // Remove a property

MACRO\_EXTEND\_WMS( DeleteLocalFld, ElementU, OpSideEffect|OpHasImplicitCall|OpDoNotTransfer|OpPostOpDbgBailOut) // Remove a property

MACRO\_WMS( DeleteRootFld, ElementC, OpSideEffect|OpHasImplicitCall|OpDoNotTransfer|OpPostOpDbgBailOut) // Remove a property (access to let/const on root object)

MACRO\_WMS( DeleteFldStrict, ElementC, OpSideEffect|OpHasImplicitCall|OpDoNotTransfer|OpPostOpDbgBailOut) // Remove a property in strict mode

MACRO\_WMS( DeleteRootFldStrict, ElementC, OpSideEffect|OpHasImplicitCall|OpDoNotTransfer|OpPostOpDbgBailOut) // Remove a property in strict mode (access to let/const on root object)

MACRO\_WMS( ScopedLdFld, ElementP, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Load from function's scope stack

MACRO\_EXTEND\_WMS( ScopedLdFldForTypeOf, ElementP, OpSideEffect|OpHasImplicitCall| OpPostOpDbgBailOut) // Load from function's scope stack for Typeof of a property

MACRO\_WMS( ScopedLdMethodFld, ElementCP, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Load call target from ScriptObject instance's direct field, but either scope object or root load from root object

MACRO\_WMS( ScopedLdInst, ElementScopedC2,OpSideEffect|OpHasImplicitCall) // Load owning instance from function's scope stack (NOTE: HasProperty may call DOM)

MACRO\_WMS( ScopedInitFunc, ElementScopedC, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Init on instance on scope stack

MACRO\_WMS( ScopedStFld, ElementP, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Store to function's scope stack

MACRO\_EXTEND\_WMS( ConsoleScopedStFld, ElementP, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Store to function's scope stack

MACRO\_WMS( ScopedStFldStrict, ElementP, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Store to function's scope stack

MACRO\_WMS( ScopedDeleteFld, ElementScopedC, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Remove a property through a stack of scopes

MACRO\_WMS( ScopedDeleteFldStrict, ElementScopedC, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Remove a property through a stack of scopes in strict mode

MACRO\_WMS\_PROFILED( LdSlot, ElementSlot, OpTempNumberSources)

MACRO\_WMS\_PROFILED( LdEnvSlot, ElementSlotI2, OpTempNumberSources)

MACRO\_WMS\_PROFILED( LdInnerSlot, ElementSlotI2, OpTempNumberSources)

MACRO\_WMS\_PROFILED( LdLocalSlot, ElementSlotI1, OpTempNumberSources)

MACRO\_BACKEND\_ONLY( LdSlotArr, ElementSlot, OpTempNumberSources)

MACRO\_EXTEND\_WMS\_AND\_PROFILED(LdInnerObjSlot, ElementSlotI2, OpTempNumberSources)

MACRO\_EXTEND\_WMS\_AND\_PROFILED(LdObjSlot, ElementSlot, None)

MACRO\_EXTEND\_WMS\_AND\_PROFILED(LdLocalObjSlot, ElementSlotI1, None)

MACRO\_EXTEND\_WMS\_AND\_PROFILED(LdEnvObjSlot, ElementSlotI2, None)

MACRO\_BACKEND\_ONLY( StSlot, ElementSlot, None)

MACRO\_WMS( StEnvSlot, ElementSlotI2, None)

MACRO\_WMS( StInnerSlot, ElementSlotI2, None)

MACRO\_WMS( StLocalSlot, ElementSlotI1, None)

MACRO\_BACKEND\_ONLY( StSlotChkUndecl, ElementSlot, OpSideEffect)

MACRO\_EXTEND\_WMS( StEnvSlotChkUndecl, ElementSlotI2, OpSideEffect)

MACRO\_EXTEND\_WMS( StInnerSlotChkUndecl, ElementSlotI2, OpSideEffect)

MACRO\_EXTEND\_WMS( StLocalSlotChkUndecl, ElementSlotI1, OpSideEffect)

MACRO\_EXTEND\_WMS( StObjSlot, ElementSlot, OpSideEffect)

MACRO\_EXTEND\_WMS( StInnerObjSlot, ElementSlotI2, OpSideEffect)

MACRO\_EXTEND\_WMS( StLocalObjSlot, ElementSlotI1, OpSideEffect)

MACRO\_EXTEND\_WMS( StLocalObjSlotChkUndecl, ElementSlotI1, OpSideEffect)

MACRO\_EXTEND\_WMS( StEnvObjSlot, ElementSlotI2, OpSideEffect)

MACRO\_EXTEND\_WMS( StObjSlotChkUndecl, ElementSlot, OpSideEffect)

MACRO\_EXTEND\_WMS( StInnerObjSlotChkUndecl, ElementSlotI2, OpSideEffect)

MACRO\_EXTEND\_WMS( StEnvObjSlotChkUndecl, ElementSlotI2, OpSideEffect)

MACRO\_BACKEND\_ONLY( LdAsmJsSlot, ElementSlot, OpTempNumberSources|OpCanCSE)

MACRO\_BACKEND\_ONLY( StAsmJsSlot, ElementSlot, None)

#ifndef FLOAT\_VAR

MACRO\_BACKEND\_ONLY( StSlotBoxTemp, Empty, OpSideEffect|OpTempNumberSources)

#endif

MACRO\_WMS\_PROFILED( LdElemI\_A, ElementI, OpHasImplicitCall|OpCanCSE|OpPostOpDbgBailOut) // Load from instance's indirect element / field, checked

MACRO\_WMS( LdMethodElem, ElementI, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Load call target from instance's indirect element / field, checked

MACRO\_WMS\_PROFILED( StElemI\_A, ElementI, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Store into instance's indirect element / field, checked

MACRO\_WMS\_PROFILED( StElemI\_A\_Strict, ElementI, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Store into instance's indirect element / field, checked

MACRO\_BACKEND\_ONLY( StInt8ArrViewElem, ElementI, OpSideEffect ) // Store Int8 into typed array view

MACRO\_BACKEND\_ONLY( StUInt8ArrViewElem, ElementI, OpSideEffect ) // Store UInt8 into typed array view

MACRO\_BACKEND\_ONLY( StInt16ArrViewElem, ElementI, OpSideEffect ) // Store Int16 into typed array view

MACRO\_BACKEND\_ONLY( StUInt16ArrViewElem, ElementI, OpSideEffect ) // Store UInt16 into typed array view

MACRO\_BACKEND\_ONLY( StInt32ArrViewElem, ElementI, OpSideEffect ) // Store Int32 into typed array view

MACRO\_BACKEND\_ONLY( StUInt32ArrViewElem, ElementI, OpSideEffect ) // Store UInt32 into typed array view

MACRO\_BACKEND\_ONLY( StFloat32ArrViewElem, ElementI, OpSideEffect ) // Store Float32 into typed array view

MACRO\_BACKEND\_ONLY( StFloat64ArrViewElem, ElementI, OpSideEffect ) // Store Float64 into typed array view

MACRO\_BACKEND\_ONLY( LdFloat32ArrViewElem, ElementI, OpCanCSE ) // load Float32 typed array view

MACRO\_BACKEND\_ONLY( LdFloat64ArrViewElem, ElementI, OpCanCSE ) // load Float64 typed array view

MACRO\_BACKEND\_ONLY( LdInt8ArrViewElem, ElementI, OpCanCSE ) // load Int8 from typed array view

MACRO\_BACKEND\_ONLY( LdInt16ArrViewElem, ElementI, OpCanCSE ) // load Int16 from typed array view

MACRO\_BACKEND\_ONLY( LdUInt8ArrViewElem, ElementI, OpCanCSE ) // load UInt8 from typed array view

MACRO\_BACKEND\_ONLY( LdUInt16ArrViewElem, ElementI, OpCanCSE ) // load UInt16 from typed array view

MACRO\_BACKEND\_ONLY( LdInt32ArrViewElem, ElementI, OpCanCSE ) // load Int32 from typed array view

MACRO\_BACKEND\_ONLY( LdUInt32ArrViewElem, ElementI, OpCanCSE ) // load UInt32 from typed array view

MACRO\_BACKEND\_ONLY( Memset, ElementI, OpSideEffect)

MACRO\_BACKEND\_ONLY( Memcopy, ElementI, OpSideEffect)

MACRO\_BACKEND\_ONLY( ArrayDetachedCheck, Reg1, None) // ensures that an ArrayBuffer has not been detached

MACRO\_WMS( StArrItemI\_CI4, ElementUnsigned1, OpSideEffect)

MACRO\_WMS( StArrItemC\_CI4, ElementUnsigned1, OpSideEffect)

MACRO\_WMS( LdArrHead, Reg2, OpTempObjectSources)

MACRO\_BACKEND\_ONLY( BoundCheck, Empty, OpTempNumberSources|OpTempObjectSources)

MACRO\_BACKEND\_ONLY( UnsignedBoundCheck, Empty, OpTempNumberSources|OpTempObjectSources)

MACRO\_WMS( StArrInlineItem\_CI4, ElementUnsigned1, OpSideEffect)

MACRO\_WMS( StArrSegItem\_CI4, ElementUnsigned1, OpSideEffect)

MACRO( StArrSegItem\_A, Auxiliary, OpSideEffect)

MACRO\_WMS( DeleteElemI\_A, ElementI, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Remove from instance's indirect element / field, checked

MACRO\_WMS( DeleteElemIStrict\_A, ElementI, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Remove from instance's indirect element / field, checked

MACRO\_EXTEND\_WMS( InitSetFld, ElementC, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut) // Set in Object Literal Syntax {set prop(args){}};

MACRO\_EXTEND\_WMS( InitGetFld, ElementC, OpSideEffect|OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut) // Get in Object Literal Syntax {get prop(){}};

MACRO\_EXTEND\_WMS( InitSetElemI, ElementI, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Set in Object Literal Syntax {set [expr](args){}};

MACRO\_EXTEND\_WMS( InitGetElemI, ElementI, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Get in Object Literal Syntax {get [expr](args){}};

MACRO\_EXTEND\_WMS( InitComputedProperty, ElementI, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Data property in Object Literal Syntax { [expr] : expr};

MACRO\_EXTEND\_WMS( InitProto, ElementC, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // Declare \_\_proto\_\_ in Object Literal Syntax {\_\_proto\_\_: ...}

MACRO\_WMS\_PROFILED( LdLen\_A, Reg2, OpHasImplicitCall|OpFastFldInstr|OpPostOpDbgBailOut|OpTempObjectSources) // Push array / string length

MACRO\_WMS( LdUndef, Reg1, OpByteCodeOnly|OpCanCSE) // Load 'undefined'

MACRO\_WMS( LdNaN, Reg1, OpByteCodeOnly|OpCanCSE) // Load 'NaN'

MACRO\_WMS( LdInfinity, Reg1, OpByteCodeOnly|OpCanCSE) // Load 'Infinity'

MACRO\_WMS( LdTrue, Reg1, OpByteCodeOnly|OpCanCSE) // Load 'true' boolean primitive

MACRO\_WMS( LdFalse, Reg1, OpByteCodeOnly|OpCanCSE) // Load 'false' boolean primitive

MACRO\_BACKEND\_ONLY( LdEnv, Reg1, OpCanCSE) // Load the optional FixedSizeArray environment for closures

MACRO\_BACKEND\_ONLY( LdAsmJsEnv, Reg1, OpCanCSE) // Load the asm.js memory

MACRO\_WMS( LdArgCnt, Reg1, None) // Load the argument count from the current function

MACRO\_BACKEND\_ONLY( InitLoopBodyCount, Reg1, None) // fake instruction for loop body counter init

MACRO\_BACKEND\_ONLY( IncrLoopBodyCount, Reg2, None) // fake instruction for loop body counter increment

MACRO\_BACKEND\_ONLY( StLoopBodyCount, Reg1, OpSideEffect) // fake instruction for loop body counter store to loopheader profiledLoopCounter

MACRO\_WMS( LdHeapArguments, Reg2, OpSideEffect) // Load the heap-based "arguments" object

MACRO\_WMS( LdLetHeapArguments, Reg2, OpSideEffect) // Load the heap-based "arguments" object (formals are let-like instead of var-like)

MACRO\_BACKEND\_ONLY( LdArgumentsFromStack,Reg1, None) // Load the heap-based "arguments" object even if it is null (Loads from meta arguments location for inlinee as well).

MACRO\_WMS( LdHeapArgsCached, Reg1, OpSideEffect) // Load the heap-based "arguments" object in a cached scope

MACRO\_EXTEND\_WMS( LdLetHeapArgsCached,Reg1, OpSideEffect) // Load the heap-based "arguments" object in a cached scope (formals are let-like instead of var-like)

MACRO\_EXTEND\_WMS( LdStackArgPtr, Reg1, OpSideEffect) // Load the address of the base of the input parameter area

MACRO\_WMS\_PROFILED\_OP( LdThis, Reg2Int1, OpHasImplicitCall|OpTempNumberTransfer) // Load this object (NOTE: TryLoadRoot exit scripts on host dispatch, but otherwise, no side effect)

MACRO\_WMS\_PROFILED\_OP( StrictLdThis, Reg2, OpHasImplicitCall|OpTempNumberTransfer) // Load this object in strict mode

MACRO\_BACKEND\_ONLY( CheckThis, Reg1, OpCanCSE|OpBailOutRec)

MACRO\_BACKEND\_ONLY( StrictCheckThis, Reg1, OpCanCSE|OpBailOutRec)

MACRO\_BACKEND\_ONLY( LdHandlerScope, Reg1, OpHasImplicitCall) // Load a scope stack for an event handler (both "this" and parent scopes)

MACRO\_BACKEND\_ONLY( LdFrameDisplay, Reg3, None) // Set up a frame display for this function and its parent frames

#if DBG

MACRO\_BACKEND\_ONLY( LdFrameDisplayNoParent,Reg1, None) // Set up a frame display for this function and its parent frames

#endif

MACRO\_WMS( LdInnerFrameDisplay,Reg3, None) // Set up a frame display for this function and its parent frames -- this is for an inner scope, not the function-level scope

MACRO\_WMS( LdInnerFrameDisplayNoParent,Reg2, None) // Set up a frame display for this function and its parent frames -- this is for an inner scope, not the function-level scope

MACRO\_WMS( LdIndexedFrameDisplay,Reg2Int1, None) // Set up a frame display for this function and its parent frames -- this is for an inner scope, not the function-level scope

MACRO\_WMS( LdIndexedFrameDisplayNoParent,Reg1Unsigned1, None) // Set up a frame display for this function and its parent frames -- this is for an inner scope, not the function-level scope

MACRO\_WMS( LdFuncExprFrameDisplay,Reg2, None)

MACRO\_BACKEND\_ONLY( NewStackFrameDisplay,Reg3, None) // Set up a frame display allocated on the stack

MACRO\_WMS( IsIn, Reg3, OpSideEffect|OpCallsValueOf|OpHasImplicitCall|OpPostOpDbgBailOut) // "x in y" (NOTE: calls valueOf for the index

MACRO\_WMS( LdArgumentsFromFrame,Reg1, None) // Load the argument object from frame

MACRO\_WMS( LdElemUndef, ElementU, OpSideEffect) // Load 'undefined' to instance.property if not already present

MACRO\_EXTEND\_WMS( LdLocalElemUndef, ElementRootU, OpSideEffect) // Load 'undefined' to instance.property if not already present

MACRO\_WMS( LdElemUndefScoped, ElementScopedU, OpSideEffect) // Load 'undefined' to [env].property if not already present in the scope

MACRO\_WMS( LdFuncExpr, Reg1, None) // Load the function expression to its location

MACRO( LdPropIds, Auxiliary, None) // Load the property id array

MACRO\_WMS( StFuncExpr, ElementC, OpSideEffect) // Store the function expression to an activation object with attributes

MACRO\_WMS( StLocalFuncExpr, ElementU, OpSideEffect) // Store the function expression to the local activation object with attributes

MACRO\_EXTEND\_WMS( LdNewTarget, Reg1, None) // Load new.target in an ordinary function call or construct call

MACRO\_EXTEND( ChkNewCallFlag, Empty, OpSideEffect) // Check to see if the current call has the CallFlags\_New flag set and throw if it does not

// Instancing operations

MACRO\_EXTEND\_WMS(LdCustomSpreadIteratorList, Reg2, OpHasImplicitCall|OpSideEffect) // Runs the iterator on a function spread Argument

MACRO\_WMS\_PROFILED\_INDEX( NewScObject, CallI, OpSideEffect|OpUseAllFields|OpCallInstr) // Create new ScriptObject instance

MACRO\_EXTEND\_WMS\_AND\_PROFILED(NewScObjectSpread, CallIExtended, OpSideEffect|OpUseAllFields|OpCallInstr) // Create new ScriptObject instance

MACRO\_WMS\_PROFILED2( NewScObjArray, CallI, OpSideEffect|OpUseAllFields|OpCallInstr) // Create new ScriptObject instance

MACRO\_WMS\_PROFILED2( NewScObjArraySpread, CallIExtended, OpSideEffect|OpUseAllFields|OpCallInstr) // Create new ScriptObject instance

MACRO( NewScObject\_A, Auxiliary, OpSideEffect|OpUseAllFields) // Create new ScriptObject instance passing only constants

MACRO\_WMS( NewScObjectNoCtorFull, Reg2, OpHasImplicitCall | OpTempObjectCanStoreTemp) // Create new object that will be used for the 'this' binding in a base class constructor

MACRO\_BACKEND\_ONLY( NewScObjectNoCtor, Empty, OpHasImplicitCall | OpTempObjectCanStoreTemp) // Create new object that will be passed into a constructor

MACRO\_BACKEND\_ONLY( GetNewScObject, Empty, OpTempObjectTransfer) // Determine which object to finally use as the result of NewScObject (object passed into constructor as 'this', or object returned by constructor)

MACRO\_BACKEND\_ONLY( UpdateNewScObjectCache, Empty, None) // Update the cache used for NewScObject

MACRO\_WMS( NewScObjectSimple, Reg1, OpTempObjectCanStoreTemp)

MACRO( NewScObjectLiteral, Auxiliary, OpSideEffect|OpTempObjectCanStoreTemp) // Create new ScriptObject instance with no constructor or arguments

MACRO\_WMS\_PROFILED( NewScArray, Reg1Unsigned1, OpSideEffect|OpTempObjectProducing) // Create new ScriptArray instance

MACRO\_WMS( NewScArrayWithMissingValues, Reg1Unsigned1, OpSideEffect|OpTempObjectProducing) // Create new ScriptArray instance

MACRO\_PROFILED( NewScIntArray, Auxiliary, OpSideEffect|OpTempObjectProducing) // Create new ScriptArray instance

MACRO\_PROFILED( NewScFltArray, Auxiliary, OpSideEffect|OpTempObjectProducing) // Create new ScriptArray instance

MACRO\_EXTEND\_WMS( InitClass, Class, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut)

MACRO\_WMS( NewScFunc, ElementSlotI1, OpSideEffect) // Create new ScriptFunction instance

MACRO\_WMS( NewScGenFunc, ElementSlotI1, OpSideEffect) // Create new JavascriptGeneratorFunction instance

MACRO\_WMS( NewStackScFunc, ElementSlotI1, OpSideEffect|OpByteCodeOnly) // Create new ScriptFunction instance

MACRO\_EXTEND\_WMS( NewInnerScFunc, ElementSlot, OpSideEffect) // Create new ScriptFunction instance

MACRO\_EXTEND\_WMS( NewInnerScGenFunc, ElementSlot, OpSideEffect) // Create new JavascriptGeneratorFunction instance

MACRO\_EXTEND\_WMS( NewInnerStackScFunc,ElementSlot, OpSideEffect|OpByteCodeOnly) // Create new ScriptFunction instance

MACRO\_BACKEND\_ONLY( NewScopeObject, Reg1, None) // Create new NewScopeObject

MACRO\_BACKEND\_ONLY( InitCachedScope, Reg2Aux, OpSideEffect) // Retrieve cached scope; create if not cached

MACRO\_BACKEND\_ONLY( InitLetCachedScope, Reg2Aux, OpSideEffect) // Retrieve cached scope; create if not cached (formals are let-like instead of var-like)

MACRO( InitCachedFuncs, AuxNoReg, OpSideEffect)

MACRO\_WMS( GetCachedFunc, Reg1Unsigned1, None)

MACRO( CommitScope, AuxNoReg, OpSideEffect) // Mark the cached scope object as committed on exit from the function

MACRO\_WMS( InvalCachedScope, Unsigned1, OpSideEffect)

MACRO\_WMS( NewPseudoScope, Unsigned1, None) // Create new scope that can't take normal var inits

MACRO\_WMS( NewBlockScope, Unsigned1, None) // Create new scope that takes only block-scoped inits

MACRO\_WMS( CloneBlockScope, Unsigned1, OpSideEffect) // Clone existing block scope in place for for-loop iterations

MACRO\_BACKEND\_ONLY( NewScopeSlots, Reg1Unsigned1, None)

MACRO\_BACKEND\_ONLY( NewStackScopeSlots, Reg1, None)

MACRO\_BACKEND\_ONLY( InitLocalClosure, Reg1, None)

MACRO\_WMS( NewInnerScopeSlots, Reg3, None)

MACRO\_WMS( CloneInnerScopeSlots, Unsigned1, OpSideEffect) // Clone existing inner scope slots in place for for-loop iterations

MACRO\_BACKEND\_ONLY( NewScopeSlotsWithoutPropIds, Reg1Int2, None)

MACRO\_WMS( NewRegEx, Reg1Unsigned1, OpTempObjectCanStoreTemp|OpSideEffect) // Create a new RegEx expression

MACRO\_WMS( IsInst, Reg3C, OpSideEffect|OpHasImplicitCall|OpPostOpDbgBailOut) // instanceof() - SideEffect: can throw...

// Collection operations

MACRO\_WMS( GetForInEnumerator, Reg2, OpSideEffect|OpHasImplicitCall) // Get enumerator from collection

MACRO\_WMS( BrOnEmpty, BrReg2, OpSideEffect|OpHasImplicitCall) // Move to next item; return value if not NULL, otherwise branch

MACRO\_BACKEND\_ONLY ( BrOnNotEmpty, BrReg2, OpSideEffect|OpHasImplicitCall) // Move to next item; return true if done

MACRO\_WMS( ReleaseForInEnumerator, Reg1, OpSideEffect|OpHasImplicitCall) // Release enumerator

MACRO( TryCatch, Br, OpSideEffect)

MACRO( TryFinally, Br, OpSideEffect|OpPostOpDbgBailOut)

MACRO\_EXTEND\_WMS( TryFinallyWithYield, BrReg2, OpSideEffect|OpPostOpDbgBailOut)

MACRO\_WMS( Catch, Reg1, OpSideEffect)

MACRO\_EXTEND( ResumeCatch, Empty, OpSideEffect)

MACRO\_EXTEND\_WMS( ResumeFinally, BrReg2, OpSideEffect)

MACRO( LeaveNull, Empty, OpSideEffect|OpNoFallThrough)

MACRO( Leave, Empty, OpSideEffect|OpNoFallThrough)

MACRO\_BACKEND\_ONLY( InlineRuntimeTypeError, W1, OpSideEffect|OpPostOpDbgBailOut) // Throws TypeError at runtime.

MACRO\_EXTEND( RuntimeTypeError, W1, OpSideEffect|OpPostOpDbgBailOut) // Throws TypeError at runtime.

MACRO\_BACKEND\_ONLY( InlineRuntimeReferenceError, W1, OpSideEffect|OpPostOpDbgBailOut) // Throws ReferenceError at runtime.

MACRO\_EXTEND( RuntimeReferenceError, W1, OpSideEffect|OpPostOpDbgBailOut) // Throws ReferenceError at runtime.

// Dynamic profile opcodes

MACRO\_WMS( LoopBodyStart, Unsigned1, OpByteCodeOnly) // Marks the start of a loop body

MACRO\_WMS( ProfiledLoopStart, Unsigned1, OpSideEffect) // Marks the start of a profiled loop

MACRO\_WMS( ProfiledLoopBodyStart, Unsigned1, OpSideEffect) // Marks the start of a profiled loop body

MACRO\_WMS( ProfiledLoopEnd, Unsigned1, OpSideEffect) // Marks the end of a profiled loop

// The order of this need to be the same as the Call\* order

MACRO\_WMS( ProfiledCallI, ProfiledCallI, OpByteCodeOnly|OpSideEffect|OpUseAllFields|OpCallInstr|OpInlineCallInstr) // Return <- Call (indirect) Function(ArgCount)

MACRO\_WMS( ProfiledCallIFlags, ProfiledCallIFlags, OpByteCodeOnly|OpSideEffect|OpUseAllFields|OpCallInstr|OpInlineCallInstr) // Return <- Call (indirect) Function(ArgCount)

MACRO\_WMS( ProfiledCallIExtended, ProfiledCallIExtended, OpByteCodeOnly|OpSideEffect|OpUseAllFields|OpCallInstr)

MACRO\_WMS( ProfiledCallIExtendedFlags, ProfiledCallIExtendedFlags, OpByteCodeOnly|OpSideEffect|OpUseAllFields| OpCallInstr)

MACRO\_WMS( ProfiledCallIWithICIndex, ProfiledCallIWithICIndex, OpByteCodeOnly|OpSideEffect|OpUseAllFields|OpCallInstr|OpInlineCallInstr) // Return <- Call (indirect) Function(ArgCount)

MACRO\_WMS( ProfiledCallIFlagsWithICIndex, ProfiledCallIFlagsWithICIndex, OpByteCodeOnly|OpSideEffect|OpUseAllFields|OpCallInstr|OpInlineCallInstr) // Return <- Call (indirect) Function(ArgCount)

MACRO\_WMS( ProfiledCallIExtendedWithICIndex, ProfiledCallIExtendedWithICIndex, OpByteCodeOnly|OpSideEffect|OpUseAllFields|OpCallInstr|OpInlineCallInstr) // Return <- Call (indirect) Function(ArgCount)

MACRO\_WMS( ProfiledCallIExtendedFlagsWithICIndex, ProfiledCallIExtendedFlagsWithICIndex, OpByteCodeOnly|OpSideEffect|OpUseAllFields|OpCallInstr)

// The order of this needs to be the same as the Call\* order

MACRO\_WMS( ProfiledReturnTypeCallI, ProfiledCallI, OpByteCodeOnly|OpSideEffect|OpUseAllFields|OpCallInstr|OpInlineCallInstr) // Return <- Call (indirect) Function(ArgCount)

MACRO\_WMS( ProfiledReturnTypeCallIFlags, ProfiledCallIFlags, OpByteCodeOnly|OpSideEffect|OpUseAllFields|OpCallInstr|OpInlineCallInstr) // Return <- Call (indirect) Function(ArgCount)

MACRO\_WMS( ProfiledReturnTypeCallIExtended, ProfiledCallIExtended, OpByteCodeOnly|OpSideEffect|OpUseAllFields|OpCallInstr)

MACRO\_WMS( ProfiledReturnTypeCallIExtendedFlags, ProfiledCallIExtendedFlags, OpByteCodeOnly|OpSideEffect|OpUseAllFields|OpCallInstr)

MACRO\_EXTEND\_WMS( EmitTmpRegCount, Unsigned1, OpByteCodeOnly)

MACRO\_WMS( Unused, Reg1, None)

// String operations

MACRO\_WMS( Concat3, Reg4, OpByteCodeOnly|OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut)

MACRO\_WMS( NewConcatStrMulti, Reg3B1, None) // Although the byte code version include the concat, and has value of/to string, the BE version doesn't

MACRO\_BACKEND\_ONLY( NewConcatStrMultiBE, Reg3B1, OpCanCSE) // Although the byte code version include the concat, and has value of/to string, the BE version doesn't

MACRO\_WMS( SetConcatStrMultiItem, Reg2B1, None) // Although the byte code version include the concat, and has value of/to string, the BE version doesn't

MACRO\_BACKEND\_ONLY( SetConcatStrMultiItemBE, Reg2B1, OpCanCSE) // Although the byte code version include the concat, and has value of/to string, the BE version doesn't

MACRO\_WMS( SetConcatStrMultiItem2, Reg3B1, None) // Although the byte code version include the concat, and has value of/to string, the BE version doesn't

MACRO\_BACKEND\_ONLY( LdStr, Empty, OpTempNumberProducing|OpCanCSE) // Load string literal

MACRO\_BACKEND\_ONLY( CloneStr, Empty, OpTempNumberSources | OpTempNumberProducing) // Load string literal

// Operation ToString(str) if str != null or str != undefined

MACRO\_BACKEND\_ONLY( Coerse\_Str, Empty, OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources|OpPostOpDbgBailOut)

// Operation CreateRegex(regex)

MACRO\_BACKEND\_ONLY( Coerse\_Regex, Empty, OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempNumberTransfer|OpPostOpDbgBailOut)

// Operation CreateRegex(regex) or CoerseStr(arg)

MACRO\_BACKEND\_ONLY( Coerse\_StrOrRegex, Empty, OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempNumberTransfer|OpPostOpDbgBailOut)

// Operation ToString(ToPrimtive(src1))

MACRO\_BACKEND\_ONLY( Conv\_PrimStr, Empty, OpCallsValueOf|OpHasImplicitCall|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpPostOpDbgBailOut)

// The following are for lowering; currently the specified layout is not used

MACRO\_BACKEND\_ONLY( StElemC, ElementC, OpSideEffect) // Store into Array instance's direct element, unchecked

MACRO\_BACKEND\_ONLY( StArrSegElemC, ElementC, OpSideEffect) // Store into Array segment's direct element, unchecked

MACRO\_EXTEND( Label, Empty, OpSideEffect) // Label (branch target)

MACRO\_BACKEND\_ONLY( CallHelper, Empty, OpSideEffect|OpUseAllFields) // Call a helper function

MACRO\_EXTEND( FunctionEntry, Empty, OpSideEffect) // Marks the start of a function

MACRO\_EXTEND( FunctionExit, Empty, OpSideEffect|OpNoFallThrough) // Marks the end of a function

MACRO\_EXTEND( StatementBoundary, Empty, None) // Marks the start or end of a statement

MACRO\_BACKEND\_ONLY( BailOut, Empty, OpSideEffect|OpBailOutRec)

MACRO\_BACKEND\_ONLY( BailOnEqual, Empty, OpBailOutRec|OpTempNumberSources|OpTempObjectSources|OpCanCSE)

MACRO\_BACKEND\_ONLY( BailOnNotEqual, Empty, OpBailOutRec|OpTempNumberSources|OpTempObjectSources|OpCanCSE)

MACRO\_BACKEND\_ONLY( BailOnNegative, Empty, OpBailOutRec|OpTempNumberSources|OpTempObjectSources|OpCanCSE)

MACRO\_BACKEND\_ONLY( BailOnNotStackArgs, Empty, OpBailOutRec|OpTempNumberSources|OpTempObjectSources|OpCanCSE) // Bail out if not stack args or actuals exceed InlineeCallInfo::MaxInlineeArgoutCount (15)

MACRO\_BACKEND\_ONLY( BailOnNotSpreadable, Empty, OpBailOutRec|OpTempNumberSources|OpTempObjectSources|OpCanCSE)

MACRO\_BACKEND\_ONLY( BailOnNotPolymorphicInlinee, Empty, OpBailOutRec|OpTempNumberSources)

MACRO\_BACKEND\_ONLY( BailTarget, Empty, OpBailOutRec|OpTempNumberSources|OpTempObjectSources)

MACRO\_BACKEND\_ONLY( BailOnNoProfile, Empty, OpBailOutRec|OpDeadFallThrough)

MACRO\_BACKEND\_ONLY( BailOnNoSimdTypeSpec, Empty, OpBailOutRec|OpDeadFallThrough)

MACRO\_BACKEND\_ONLY( BailOnNotObject, Empty, OpBailOutRec|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpTempObjectSources)

MACRO\_BACKEND\_ONLY( BailOnNotArray, Empty, OpBailOutRec|OpTempNumberSources|OpTempObjectSources|OpCanCSE|OpTempObjectSources)

MACRO\_BACKEND\_ONLY( BailForDebugger, Empty, OpBailOutRec|OpTempNumberSources|OpTempObjectSources|OpSideEffect) // Bail out so that we can continue the function under debugger. Disable optimizations for this instr so that it's not moved.

MACRO\_BACKEND\_ONLY( BailOnNotBuiltIn, Empty, OpBailOutRec|OpTempNumberSources|OpTempObjectSources|OpCanCSE)

MACRO\_BACKEND\_ONLY( BailOnException, Empty, OpBailOutRec|OpTempNumberSources|OpTempObjectSources|OpDeadFallThrough)

MACRO\_BACKEND\_ONLY( BailOnTaggedValue, Empty, OpBailOutRec|OpTempNumberSources|OpTempObjectSources|OpCanCSE)

MACRO\_BACKEND\_ONLY( BytecodeArgOutCapture, Empty, OpTempNumberTransfer|OpTempObjectTransfer|OpNonIntTransfer) // Represents snapshotting of bytecode ArgOut\_A in backend for purpose of bailout

MACRO\_BACKEND\_ONLY( BytecodeArgOutUse, Empty, OpTempNumberSources | OpTempObjectSources) // Represents bytecode ArgOut\_A use in the backend to keep args alive for the globopt

MACRO\_BACKEND\_ONLY( ByteCodeUses, Empty, OpSideEffect|OpTempNumberSources|OpTempObjectSources)

MACRO\_BACKEND\_ONLY( BailOutStackRestore,Empty, OpSideEffect)

MACRO\_BACKEND\_ONLY( NoImplicitCallUses, Empty, None)

MACRO\_BACKEND\_ONLY( NoIntOverflowBoundary, Empty, None)

MACRO\_BACKEND\_ONLY( EndCallForPolymorphicInlinee, Empty, OpSideEffect|OpDeadFallThrough)

MACRO\_BACKEND\_ONLY( InlineeStart, Empty, OpSideEffect)

MACRO\_BACKEND\_ONLY( InlineeEnd, Empty, OpSideEffect)

MACRO\_BACKEND\_ONLY( InlineeMetaArg, Empty, OpSideEffect)

MACRO\_BACKEND\_ONLY( InlineBuiltInStart, Empty, OpSideEffect)

MACRO\_BACKEND\_ONLY( InlineBuiltInEnd, Empty, OpSideEffect)

MACRO\_BACKEND\_ONLY( InlineNonTrackingBuiltInEnd, Empty, OpSideEffect)

MACRO\_BACKEND\_ONLY( InlineMathAcos, Empty, OpInlinableBuiltIn|OpTempNumberSources|OpProducesNumber|OpCanCSE)

MACRO\_BACKEND\_ONLY( InlineMathAsin, Empty, OpInlinableBuiltIn|OpTempNumberSources|OpProducesNumber|OpCanCSE)

MACRO\_BACKEND\_ONLY( InlineMathAtan, Empty, OpInlinableBuiltIn|OpTempNumberSources|OpProducesNumber|OpCanCSE)

MACRO\_BACKEND\_ONLY( InlineMathAtan2, Empty, OpInlinableBuiltIn|OpTempNumberSources|OpProducesNumber|OpCanCSE)

MACRO\_BACKEND\_ONLY( InlineMathCos, Empty, OpInlinableBuiltIn|OpTempNumberSources|OpProducesNumber|OpCanCSE)

MACRO\_BACKEND\_ONLY( InlineMathExp, Empty, OpInlinableBuiltIn|OpTempNumberSources|OpProducesNumber|OpCanCSE)

MACRO\_BACKEND\_ONLY( InlineMathLog, Empty, OpInlinableBuiltIn|OpTempNumberSources|OpProducesNumber|OpCanCSE)

MACRO\_BACKEND\_ONLY( InlineMathPow, Empty, OpInlinableBuiltIn|OpTempNumberSources|OpProducesNumber|OpCanCSE)

MACRO\_BACKEND\_ONLY( InlineMathSin, Empty, OpInlinableBuiltIn|OpTempNumberSources|OpProducesNumber|OpCanCSE)

MACRO\_BACKEND\_ONLY( InlineMathSqrt, Empty, OpInlinableBuiltIn|OpTempNumberSources|OpProducesNumber|OpCanCSE)

MACRO\_BACKEND\_ONLY( InlineMathTan, Empty, OpInlinableBuiltIn|OpTempNumberSources|OpProducesNumber|OpCanCSE)

// IE11 inline built-ins

// TODO: put these upfront so that all built-ins are sorted.

MACRO\_BACKEND\_ONLY( InlineMathAbs, Empty, OpInlinableBuiltIn|OpTempNumberSources|OpCanCSE|OpProducesNumber)

MACRO\_BACKEND\_ONLY( InlineMathClz32, Empty, OpInlinableBuiltIn|OpTempNumberSources|OpCanCSE|OpProducesNumber)

MACRO\_BACKEND\_ONLY( InlineMathCeil, Empty, OpInlinableBuiltIn|OpTempNumberSources|OpCanCSE|OpBailOutRec|OpProducesNumber)

MACRO\_BACKEND\_ONLY( InlineMathFloor, Empty, OpInlinableBuiltIn|OpTempNumberSources|OpCanCSE|OpBailOutRec|OpProducesNumber)

MACRO\_BACKEND\_ONLY( InlineMathMax, Empty, OpInlinableBuiltIn|OpTempNumberSources|OpCanCSE|OpProducesNumber)

MACRO\_BACKEND\_ONLY( InlineMathImul, Empty, OpInlinableBuiltIn|OpTempNumberSources|OpCanCSE|OpProducesNumber)

MACRO\_BACKEND\_ONLY( InlineMathMin, Empty, OpInlinableBuiltIn|OpTempNumberSources|OpCanCSE|OpProducesNumber)

MACRO\_BACKEND\_ONLY( InlineMathRandom, Empty, OpSideEffect|OpInlinableBuiltIn|OpProducesNumber)

MACRO\_BACKEND\_ONLY( InlineMathRound, Empty, OpInlinableBuiltIn|OpTempNumberSources|OpCanCSE|OpBailOutRec|OpProducesNumber)

MACRO\_BACKEND\_ONLY( InlineMathFround, Empty, OpInlinableBuiltIn|OpTempNumberSources|OpCanCSE|OpProducesNumber)

MACRO\_BACKEND\_ONLY( InlineStringCharAt, Empty, OpInlinableBuiltIn|OpCanCSE)

MACRO\_BACKEND\_ONLY( InlineStringCharCodeAt, Empty, OpInlinableBuiltIn|OpCanCSE)

MACRO\_BACKEND\_ONLY( InlineStringCodePointAt, Empty, OpInlinableBuiltIn|OpCanCSE)

MACRO\_BACKEND\_ONLY( InlineArrayPop, Empty, OpSideEffect|OpInlinableBuiltIn|OpHasImplicitCall)

MACRO\_BACKEND\_ONLY( InlineArrayPush, Empty, OpSideEffect|OpInlinableBuiltIn|OpHasImplicitCall)

MACRO\_BACKEND\_ONLY( InlineFunctionApply, Empty, OpSideEffect|OpInlinableBuiltIn)

MACRO\_BACKEND\_ONLY( InlineFunctionCall, Empty, OpSideEffect|OpInlinableBuiltIn)

MACRO\_BACKEND\_ONLY( InlineRegExpExec, Empty, OpSideEffect|OpInlinableBuiltIn)

MACRO\_BACKEND\_ONLY( CallIFixed, Empty, OpSideEffect|OpUseAllFields|OpCallInstr|OpInlineCallInstr)

MACRO\_BACKEND\_ONLY( CheckFixedFld, Empty, OpFastFldInstr|OpTempObjectSources|OpCanCSE)

MACRO\_BACKEND\_ONLY( CheckPropertyGuardAndLoadType, Empty, OpFastFldInstr|OpTempObjectSources|OpDoNotTransfer)

MACRO\_BACKEND\_ONLY( CheckObjType, Empty, OpFastFldInstr|OpTempObjectSources)

MACRO\_BACKEND\_ONLY( AdjustObjType, Empty, OpSideEffect)

// Edge inline built-ins

#ifdef ENABLE\_DOM\_FAST\_PATH

MACRO\_BACKEND\_ONLY( DOMFastPathGetter, Empty, OpCanCSE) // unlike other builtins, we don't know the return type

MACRO\_BACKEND\_ONLY( DOMFastPathSetter, Empty, OpSideEffect)

#endif

MACRO\_BACKEND\_ONLY( ExtendArg\_A, Empty, OpCanCSE) // extend a list of arguments to be passed to functions.

#ifdef \_M\_X64

MACRO\_BACKEND\_ONLY( PrologStart, Empty, None)

MACRO\_BACKEND\_ONLY( PrologEnd, Empty, None)

MACRO\_BACKEND\_ONLY( LdArgSize, Empty, None)

MACRO\_BACKEND\_ONLY( LdSpillSize, Empty, None)

#endif

#if DBG\_DUMP

MACRO\_BACKEND\_ONLY( LdRoot, Empty, None) // Load root object

MACRO\_BACKEND\_ONLY( LdModuleRoot, Empty, None) // Load module root with given moduleID

MACRO\_BACKEND\_ONLY( LdNullDisplay, Empty, None) // Load the null frame display

MACRO\_BACKEND\_ONLY( LdStrictNullDisplay,Empty, None) // Load the strict null frame display

#endif

MACRO( SpreadArrayLiteral, Reg2Aux, OpSideEffect|OpHasImplicitCall)

MACRO\_BACKEND\_ONLY( LdSpreadIndices, Empty, None)

MACRO\_EXTEND\_WMS( ClearAttributes, ElementU, None)

MACRO\_EXTEND\_WMS( ObjectFreeze, Reg1, None)

MACRO\_EXTEND\_WMS( LdSuper, Reg1, OpSideEffect)

MACRO\_EXTEND\_WMS( LdSuperCtor, Reg1, OpSideEffect)

MACRO\_EXTEND\_WMS( ScopedLdSuper, Reg1, OpSideEffect)

MACRO\_EXTEND\_WMS( ScopedLdSuperCtor, Reg1, OpSideEffect)

MACRO\_EXTEND\_WMS( SetHomeObj, Reg2, OpSideEffect)

MACRO\_BACKEND\_ONLY( BrFncCachedScopeEq, Reg2, None)

MACRO\_BACKEND\_ONLY( BrFncCachedScopeNeq,Reg2, None)

MACRO\_BACKEND\_ONLY( GeneratorResumeJumpTable, Reg1, OpSideEffect)

MACRO\_BACKEND\_ONLY( RestoreOutParam, Empty, None)

MACRO\_BACKEND\_ONLY( SlotArrayCheck, Empty, OpCanCSE)

MACRO\_BACKEND\_ONLY( FrameDisplayCheck, Empty, OpCanCSE)

// All SIMD ops are backend only for non-asmjs.

#define MACRO\_SIMD(opcode, asmjsLayout, opCodeAttrAsmJs, OpCodeAttr, ...) MACRO\_BACKEND\_ONLY(opcode, Empty, OpCodeAttr)

#define MACRO\_SIMD\_WMS(opcode, asmjsLayout, opCodeAttrAsmJs, OpCodeAttr, ...) MACRO\_BACKEND\_ONLY(opcode, Empty, OpCodeAttr)

#define MACRO\_SIMD\_BACKEND\_ONLY(opcode, asmjsLayout, opCodeAttrAsmJs, OpCodeAttr) MACRO\_BACKEND\_ONLY(opcode, Empty, OpCodeAttr)

#define MACRO\_SIMD\_EXTEND(opcode, asmjsLayout, opCodeAttrAsmJs, OpCodeAttr, ...) MACRO\_BACKEND\_ONLY(opcode, Empty, OpCodeAttr)

#define MACRO\_SIMD\_EXTEND\_WMS(opcode, asmjsLayout, opCodeAttrAsmJs, OpCodeAttr, ...) MACRO\_BACKEND\_ONLY(opcode, Empty, OpCodeAttr)

#define MACRO\_SIMD\_BACKEND\_ONLY\_EXTEND(opcode, asmjsLayout, opCodeAttrAsmJs, OpCodeAttr) MACRO\_BACKEND\_ONLY(opcode, Empty, OpCodeAttr)

#include "OpCodesSimd.h"

#undef MACRO\_WMS\_PROFILED

#undef MACRO\_WMS\_PROFILED2

#undef MACRO\_WMS\_PROFILED\_OPCODE

#undef MACRO\_PROFILED

#undef MACRO\_DEBUG\_WMS

#undef MACRO\_DEBUG

// help the caller to undefine all the macros

#undef MACRO

#undef MACRO\_WMS

#undef MACRO\_EXTEND

#undef MACRO\_EXTEND\_WMS

#undef MACRO\_BACKEND\_ONLY

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

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

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

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

// Default all macro to nothing

#ifndef MACRO

#define MACRO( opcode, layout, attr)

#endif

#ifndef MACRO\_WMS

#define MACRO\_WMS(opcode, layout, attr)

#endif

#ifndef MACRO\_EXTEND

#define MACRO\_EXTEND(opcode, layout, attr)

#endif

#ifndef MACRO\_EXTEND\_WMS

#define MACRO\_EXTEND\_WMS(opcode, layout, attr)

#endif

// ( OpCodeAsmJs , LayoutAsmJs , OpCodeAttrAsmJs )

// ( | , | , | )

// ( | , | , | )

// ( V , V , V )

MACRO ( EndOfBlock , Empty , None ) // End-of-buffer

// Prefix, order must be maintained to be the same as in OpCodes.h

MACRO ( ExtendedOpcodePrefix , Empty , None )

MACRO ( MediumLayoutPrefix , Empty , None )

MACRO ( ExtendedMediumLayoutPrefix, Empty , None )

MACRO ( LargeLayoutPrefix , Empty , None )

MACRO ( ExtendedLargeLayoutPrefix , Empty , None )

MACRO ( Nop , Empty , None ) // No operation (Default value = 0)

MACRO\_EXTEND( NopEx , Empty , None ) // No operation (Default value = 0)

MACRO ( Label , Empty , None ) // No operation (Default value = 0)

// These need to exist for the interpreterLoop.inl to compile

MACRO ( Yield , Empty , None )

MACRO ( Leave , Empty , None )

MACRO ( LeaveNull , Empty , None )

MACRO ( FinallyEnd , Empty , None )

MACRO ( Break , Empty , None )

MACRO ( Ret , Empty , None )

// External Function calls

MACRO ( StartCall , StartCall , None ) // Initialize memory for a call

MACRO\_WMS ( Call , AsmCall , None ) // Execute call and place return value in register

MACRO\_WMS ( ArgOut\_Db , Reg1Double1 , None ) // convert double to var and place it for function call

MACRO\_WMS ( ArgOut\_Int , Reg1Int1 , None ) // convert int to var and place it for function call

MACRO\_WMS ( Conv\_VTD , Double1Reg1 , None ) // convert var to double

MACRO\_WMS ( Conv\_VTI , Int1Reg1 , None ) // convert var to int

MACRO\_WMS ( Conv\_VTF , Float1Reg1 , None ) // convert var to float

// Internal calls

MACRO ( I\_StartCall , StartCall , None ) // Initialize memory for a call

MACRO\_WMS ( I\_Call , AsmCall , None ) // Execute call and place return value in register

MACRO\_WMS ( I\_ArgOut\_Db , Reg1Double1 , None ) // convert double to var and place it for function call

MACRO\_WMS ( I\_ArgOut\_Int , Reg1Int1 , None ) // convert int to var and place it for function call

MACRO\_WMS ( I\_ArgOut\_Flt , Reg1Float1 , None ) // convert float to var and place it for function call

MACRO\_WMS ( I\_Conv\_VTD , Double2 , None ) // convert var to double

MACRO\_WMS ( I\_Conv\_VTI , Int2 , None ) // convert var to int

MACRO\_WMS ( I\_Conv\_VTF , Float2 , None ) // convert var to float

// loop

MACRO\_WMS ( AsmJsLoopBodyStart , AsmUnsigned1 , None ) // Marks the start of a loop body

// Branching

MACRO ( AsmBr , AsmBr , OpNoFallThrough ) // Unconditional branch

MACRO\_WMS ( BrTrue\_Int , BrInt1 , None ) // Jumps to offset if int value is not 0

MACRO\_WMS ( BrEq\_Int , BrInt2 , None ) // Jumps to offset if both int are equals

// Switching

MACRO\_WMS ( BeginSwitch\_Int , Int2 , None ) // Start of an integer switch statement, same function as Ld\_Int

MACRO ( EndSwitch\_Int , AsmBr , OpNoFallThrough ) // End of an integer switch statement, jumps to default case or past end of switch

MACRO\_WMS ( Case\_Int , BrInt2 , None ) // Integer branch, same function as BrInt2

// Type conversion

MACRO\_WMS ( Conv\_DTI , Int1Double1 , None ) // convert double to int

MACRO\_WMS ( Conv\_FTI , Int1Float1 , None ) // convert float to int

MACRO\_WMS ( Conv\_ITD , Double1Int1 , None ) // convert int to double

MACRO\_WMS ( Conv\_FTD , Double1Float1, None ) // convert float to double

MACRO\_WMS ( Conv\_UTD , Double1Int1 , None ) // convert unsigned int to double

MACRO\_WMS ( Return\_Db , Double2 , None ) // convert double to var

MACRO\_WMS ( Return\_Flt , Float2 , None ) // convert float to var

MACRO\_WMS ( Return\_Int , Int2 , None ) // convert int to var

// Module memory manipulation

MACRO\_WMS ( LdUndef , AsmReg1 , None ) // Load 'undefined' usually in return register

MACRO\_WMS ( LdSlotArr , ElementSlot , None ) // Loads an array of Var from an array of Var

MACRO\_WMS ( LdSlot , ElementSlot , None ) // Loads a Var from an array of Var

MACRO\_WMS ( LdSlot\_Db , ElementSlot , None ) // Loads a double from the Module

MACRO\_WMS ( LdSlot\_Int , ElementSlot , None ) // Loads an Int from the Module

MACRO\_WMS ( LdSlot\_Flt , ElementSlot , None ) // Loads an Float from the Module

MACRO\_WMS ( StSlot\_Db , ElementSlot , None ) // Sets a double in the Module

MACRO\_WMS ( StSlot\_Int , ElementSlot , None ) // Sets an Int in the Module

MACRO\_WMS ( StSlot\_Flt , ElementSlot , None ) // Sets an Int in the Module

MACRO\_WMS ( LdArr\_Func , ElementSlot , None ) // opcode to load func from function tables

// Array Buffer manipulations

MACRO\_WMS ( LdArr , AsmTypedArr , None )

MACRO\_WMS ( LdArrConst , AsmTypedArr , None )

MACRO\_WMS ( StArr , AsmTypedArr , None )

MACRO\_WMS ( StArrConst , AsmTypedArr , None )

// Int math

MACRO\_WMS ( Ld\_IntConst , Int1Const1 , None ) // Sets an int register from a const int

MACRO\_WMS ( Ld\_Int , Int2 , None ) // Sets an int from another int register

MACRO\_WMS ( Neg\_Int , Int2 , None ) // int unary '-'

MACRO\_WMS ( Not\_Int , Int2 , None ) // int unary '~'

MACRO\_WMS ( LogNot\_Int , Int2 , None ) // int unary '!'

MACRO\_WMS ( Conv\_ITB , Int2 , None ) // int unary '!!' transform an int into a bool (0|1)

MACRO\_WMS ( Add\_Int , Int3 , None ) // int32 Arithmetic '+'

MACRO\_WMS ( Sub\_Int , Int3 , None ) // int32 Arithmetic '-' (subtract)

MACRO\_WMS ( Mul\_Int , Int3 , None ) // int32 Arithmetic '\*'

MACRO\_WMS ( Div\_Int , Int3 , None ) // int32 Arithmetic '/'

MACRO\_WMS ( Rem\_Int , Int3 , None ) // int32 Arithmetic '%'

MACRO\_WMS ( And\_Int , Int3 , None ) // int32 Bitwise '&'

MACRO\_WMS ( Or\_Int , Int3 , None ) // int32 Bitwise '|'

MACRO\_WMS ( Xor\_Int , Int3 , None ) // int32 Bitwise '^'

MACRO\_WMS ( Shl\_Int , Int3 , None ) // int32 Shift '<<' (signed, truncate)

MACRO\_WMS ( Shr\_Int , Int3 , None ) // int32 Shift '>>' (signed, truncate)

MACRO\_WMS ( ShrU\_Int , Int3 , None ) // int32 Shift '>>>'(unsigned, truncate)

// Unsigned int math

MACRO\_WMS ( Mul\_UInt , Int3 , None ) // unsigned int32 Arithmetic '\*'

MACRO\_WMS ( Div\_UInt , Int3 , None ) // unsigned int32 Arithmetic '/'

MACRO\_WMS ( Rem\_UInt , Int3 , None ) // unsigned int32 Arithmetic '%'

// Double math

MACRO\_WMS ( Ld\_Db , Double2 , None ) // Sets a double from another double register

MACRO\_WMS ( Neg\_Db , Double2 , None ) // Double Unary '-'

MACRO\_WMS ( Add\_Db , Double3 , None ) // Double Arithmetic '+'

MACRO\_WMS ( Sub\_Db , Double3 , None ) // Double Arithmetic '-' (subtract)

MACRO\_WMS ( Mul\_Db , Double3 , None ) // Double Arithmetic '\*'

MACRO\_WMS ( Div\_Db , Double3 , None ) // Double Arithmetic '/'

MACRO\_WMS ( Rem\_Db , Double3 , None ) // Double Arithmetic '%'

// float math

MACRO\_WMS ( Ld\_Flt , Float2 , None ) // Sets a float from another float register

MACRO\_WMS ( Neg\_Flt , Float2 , None ) // Float Unary '-'

MACRO\_WMS ( Add\_Flt , Float3 , None ) // Float Arithmetic '+'

MACRO\_WMS ( Sub\_Flt , Float3 , None ) // Float Arithmetic '-' (subtract)

MACRO\_WMS ( Mul\_Flt , Float3 , None ) // Float Arithmetic '\*'

MACRO\_WMS ( Div\_Flt , Float3 , None ) // Float Arithmetic '/'

// Int comparisons

MACRO\_WMS ( CmLt\_Int , Int3 , None ) // int32 Comparison <

MACRO\_WMS ( CmLe\_Int , Int3 , None ) // int32 Comparison <=

MACRO\_WMS ( CmGt\_Int , Int3 , None ) // int32 Comparison >

MACRO\_WMS ( CmGe\_Int , Int3 , None ) // int32 Comparison >=

MACRO\_WMS ( CmEq\_Int , Int3 , None ) // int32 Comparison ==

MACRO\_WMS ( CmNe\_Int , Int3 , None ) // int32 Comparison !=

// Unsigned int comparisons

MACRO\_WMS ( CmLt\_UnInt , Int3 , None ) // unsigned int32 Comparison <

MACRO\_WMS ( CmLe\_UnInt , Int3 , None ) // unsigned int32 Comparison <=

MACRO\_WMS ( CmGt\_UnInt , Int3 , None ) // unsigned int32 Comparison >

MACRO\_WMS ( CmGe\_UnInt , Int3 , None ) // unsigned int32 Comparison >=

// Double comparisons

MACRO\_WMS ( CmLt\_Db , Int1Double2 , None ) // double Comparison <

MACRO\_WMS ( CmLe\_Db , Int1Double2 , None ) // double Comparison <=

MACRO\_WMS ( CmGt\_Db , Int1Double2 , None ) // double Comparison >

MACRO\_WMS ( CmGe\_Db , Int1Double2 , None ) // double Comparison >=

MACRO\_WMS ( CmEq\_Db , Int1Double2 , None ) // double Comparison ==

MACRO\_WMS ( CmNe\_Db , Int1Double2 , None ) // double Comparison !=

// Float comparisons

MACRO\_WMS ( CmLt\_Flt , Int1Float2 , None ) // float Comparison <

MACRO\_WMS ( CmLe\_Flt , Int1Float2 , None ) // float Comparison <=

MACRO\_WMS ( CmGt\_Flt , Int1Float2 , None ) // float Comparison >

MACRO\_WMS ( CmGe\_Flt , Int1Float2 , None ) // float Comparison >=

MACRO\_WMS ( CmEq\_Flt , Int1Float2 , None ) // float Comparison ==

MACRO\_WMS ( CmNe\_Flt , Int1Float2 , None ) // float Comparison !=

// Math builtin functions for ints

MACRO\_WMS ( Abs\_Int , Int2 , None )

MACRO\_WMS ( Min\_Int , Int3 , None )

MACRO\_WMS ( Max\_Int , Int3 , None )

MACRO\_WMS ( Imul\_Int , Int3 , None )

MACRO\_WMS ( Clz32\_Int , Int2 , None )

// Math builtin functions for doubles & floats

MACRO\_WMS ( Sin\_Db , Double2 , None )

MACRO\_WMS ( Cos\_Db , Double2 , None )

MACRO\_WMS ( Tan\_Db , Double2 , None )

MACRO\_WMS ( Asin\_Db , Double2 , None )

MACRO\_WMS ( Acos\_Db , Double2 , None )

MACRO\_WMS ( Atan\_Db , Double2 , None )

MACRO\_WMS ( Abs\_Db , Double2 , None )

MACRO\_WMS ( Ceil\_Db , Double2 , None )

MACRO\_WMS ( Ceil\_Flt , Float2 , None )

MACRO\_WMS ( Floor\_Db , Double2 , None )

MACRO\_WMS ( Floor\_Flt , Float2 , None )

MACRO\_WMS ( Exp\_Db , Double2 , None )

MACRO\_WMS ( Log\_Db , Double2 , None )

MACRO\_WMS ( Pow\_Db , Double3 , None )

MACRO\_WMS ( Sqrt\_Db , Double2 , None )

MACRO\_WMS ( Sqrt\_Flt , Float2 , None )

MACRO\_WMS ( Abs\_Flt , Float2 , None )

MACRO\_WMS ( Atan2\_Db , Double3 , None )

MACRO\_WMS ( Min\_Db , Double3 , None )

MACRO\_WMS ( Max\_Db , Double3 , None )

// Fround

MACRO\_WMS ( Fround\_Flt , Float2 , None )

MACRO\_WMS ( Fround\_Db , Float1Double1 , None )

MACRO\_WMS ( Fround\_Int , Float1Int1 , None )

#define MACRO\_SIMD(opcode, asmjsLayout, opCodeAttrAsmJs, OpCodeAttr, ...) MACRO(opcode, asmjsLayout, opCodeAttrAsmJs)

#define MACRO\_SIMD\_WMS(opcode, asmjsLayout, opCodeAttrAsmJs, OpCodeAttr, ...) MACRO\_WMS(opcode, asmjsLayout, opCodeAttrAsmJs)

#define MACRO\_SIMD\_ASMJS\_ONLY\_WMS(opcode, asmjsLayout, opCodeAttrAsmJs, OpCodeAttr, ...) MACRO\_WMS(opcode, asmjsLayout, opCodeAttrAsmJs)

#define MACRO\_SIMD\_EXTEND(opcode, asmjsLayout, opCodeAttrAsmJs, OpCodeAttr, ...) MACRO\_EXTEND(opcode, asmjsLayout, opCodeAttrAsmJs)

#define MACRO\_SIMD\_EXTEND\_WMS(opcode, asmjsLayout, opCodeAttrAsmJs, OpCodeAttr, ...) MACRO\_EXTEND\_WMS(opcode, asmjsLayout, opCodeAttrAsmJs)

#define MACRO\_SIMD\_ASMJS\_ONLY\_EXTEND\_WMS(opcode, asmjsLayout, opCodeAttrAsmJs, OpCodeAttr, ...) MACRO\_EXTEND\_WMS(opcode, asmjsLayout, opCodeAttrAsmJs)

#include "OpCodesSimd.h"

// help the caller to undefine all the macros

#undef MACRO

#undef MACRO\_WMS

#undef MACRO\_EXTEND

#undef MACRO\_EXTEND\_WMS

#undef MACRO\_BACKEND\_ONLY

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

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

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

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

/\*

SIMD.js opcodes

- All opcodes are typed.

- Used as bytecode for AsmJs Interpreter.

- Used as IR by the backend only for both AsmJs and non-AsmJs code.

\*/

// used as both AsmJs bytecode and IR

#ifndef MACRO\_SIMD

#define MACRO\_SIMD(opcode, asmjsLayout, opCodeAttrAsmJs, OpCodeAttr, ...)

#endif

#ifndef MACRO\_SIMD\_WMS

#define MACRO\_SIMD\_WMS(opcode, asmjsLayout, opCodeAttrAsmJs, OpCodeAttr, ...)

#endif

// used as AsmJs bytecode only

#ifndef MACRO\_SIMD\_ASMJS\_ONLY\_WMS

#define MACRO\_SIMD\_ASMJS\_ONLY\_WMS(opcode, asmjsLayout, opCodeAttrAsmJs, OpCodeAttr, ...)

#endif

// used as IR only

#ifndef MACRO\_SIMD\_BACKEND\_ONLY

#define MACRO\_SIMD\_BACKEND\_ONLY(opcode, asmjsLayout, opCodeAttrAsmJs, OpCodeAttr)

#endif

// same as above but with extended opcodes

#ifndef MACRO\_SIMD\_EXTEND

#define MACRO\_SIMD\_EXTEND(opcode, asmjsLayout, opCodeAttrAsmJs, OpCodeAttr, ...)

#endif

#ifndef MACRO\_SIMD\_EXTEND\_WMS

#define MACRO\_SIMD\_EXTEND\_WMS(opcode, asmjsLayout, opCodeAttrAsmJs, OpCodeAttr, ...)

#endif

// used as AsmJs bytecode only

#ifndef MACRO\_SIMD\_ASMJS\_ONLY\_EXTEND\_WMS

#define MACRO\_SIMD\_ASMJS\_ONLY\_EXTEND\_WMS(opcode, asmjsLayout, opCodeAttrAsmJs, OpCodeAttr, ...)

#endif

// used as IR only

#ifndef MACRO\_SIMD\_BACKEND\_ONLY\_EXTEND

#define MACRO\_SIMD\_BACKEND\_ONLY\_EXTEND(opcode, asmjsLayout, opCodeAttrAsmJs, OpCodeAttr)

#endif

// OpCode , LayoutAsmJs , OpCodeAttrAsmJs, OpCodeAttr Addition macro args FuncInfo Ret and Args ValueTypes

// | | | | | | |

// v v v v v v v

MACRO\_SIMD ( Simd128\_Start , Empty , None , None , 0) // Just a marker to indicate SIMD opcodes region

// Int32x4

MACRO\_SIMD\_WMS ( Simd128\_IntsToI4 , Int32x4\_1Int4 , None , OpCanCSE , 6, &Js::SIMDInt32x4Lib::EntryInfo::Int32x4, ValueType::GetSimd128(ObjectType::Simd128Int32x4), ValueType::GetInt(false), ValueType::GetInt(false), ValueType::GetInt(false), ValueType::GetInt(false))

MACRO\_SIMD\_WMS ( Simd128\_Splat\_I4 , Int32x4\_1Int1 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_FromFloat64x2\_I4 , Int32x4\_1Float64x2\_1 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_FromFloat64x2Bits\_I4 , Int32x4\_1Float64x2\_1 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_FromFloat32x4\_I4 , Int32x4\_1Float32x4\_1 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_FromFloat32x4Bits\_I4 , Int32x4\_1Float32x4\_1 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Neg\_I4 , Int32x4\_2 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Add\_I4 , Int32x4\_3 , None , OpCanCSE , 4, &Js::SIMDInt32x4Lib::EntryInfo::Add, ValueType::GetSimd128(ObjectType::Simd128Int32x4), ValueType::GetSimd128(ObjectType::Simd128Int32x4), ValueType::GetSimd128(ObjectType::Simd128Int32x4))

MACRO\_SIMD\_WMS ( Simd128\_Sub\_I4 , Int32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Mul\_I4 , Int32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Lt\_I4 , Int32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Gt\_I4 , Int32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Eq\_I4 , Int32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Select\_I4 , Int32x4\_4 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_And\_I4 , Int32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Or\_I4 , Int32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Xor\_I4 , Int32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Not\_I4 , Int32x4\_2 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Shr\_I4 , Int32x4\_2Int1 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_ShrA\_I4 , Int32x4\_2Int1 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Shl\_I4 , Int32x4\_2Int1 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Swizzle\_I4 , Int32x4\_2Int4 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Shuffle\_I4 , Int32x4\_3Int4 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_LdSignMask\_I4 , Int1Int32x4\_1 , None , OpCanCSE , 0)

MACRO\_SIMD\_ASMJS\_ONLY\_WMS ( Simd128\_Ld\_I4 , Int32x4\_2 , None , None )

MACRO\_SIMD\_ASMJS\_ONLY\_WMS ( Simd128\_LdSlot\_I4 , ElementSlot , None , None )

MACRO\_SIMD\_ASMJS\_ONLY\_WMS ( Simd128\_StSlot\_I4 , ElementSlot , None , None )

MACRO\_SIMD\_ASMJS\_ONLY\_WMS ( Simd128\_Return\_I4 , Int32x4\_2 , None , None )

MACRO\_SIMD\_ASMJS\_ONLY\_WMS ( Simd128\_I\_ArgOut\_I4 , Reg1Int32x4\_1 , None , None )

MACRO\_SIMD\_ASMJS\_ONLY\_WMS ( Simd128\_I\_Conv\_VTI4 , Int32x4\_2 , None , None )

// Float32x4

MACRO\_SIMD\_WMS ( Simd128\_FloatsToF4 , Float32x4\_1Float4 , None , OpCanCSE , 6, &Js::SIMDFloat32x4Lib::EntryInfo::Float32x4, ValueType::GetSimd128(ObjectType::Simd128Float32x4), ValueType::Float, ValueType::Float, ValueType::Float, ValueType::Float)

MACRO\_SIMD\_WMS ( Simd128\_Splat\_F4 , Float32x4\_1Float1 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_FromFloat64x2\_F4 , Float32x4\_1Float64x2\_1 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_FromFloat64x2Bits\_F4 , Float32x4\_1Float64x2\_1 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_FromInt32x4\_F4 , Float32x4\_1Int32x4\_1 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_FromInt32x4Bits\_F4 , Float32x4\_1Int32x4\_1 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Abs\_F4 , Float32x4\_2 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Neg\_F4 , Float32x4\_2 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Add\_F4 , Float32x4\_3 , None , OpCanCSE , 4, &Js::SIMDFloat32x4Lib::EntryInfo::Add, ValueType::GetSimd128(ObjectType::Simd128Float32x4), ValueType::GetSimd128(ObjectType::Simd128Float32x4), ValueType::GetSimd128(ObjectType::Simd128Float32x4))

MACRO\_SIMD\_WMS ( Simd128\_Sub\_F4 , Float32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Mul\_F4 , Float32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Div\_F4 , Float32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Clamp\_F4 , Float32x4\_4 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Min\_F4 , Float32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Max\_F4 , Float32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Rcp\_F4 , Float32x4\_2 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_RcpSqrt\_F4 , Float32x4\_2 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Sqrt\_F4 , Float32x4\_2 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Swizzle\_F4 , Float32x4\_2Int4 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Shuffle\_F4 , Float32x4\_3Int4 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Lt\_F4 , Float32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_LtEq\_F4 , Float32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Eq\_F4 , Float32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Neq\_F4 , Float32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Gt\_F4 , Float32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_GtEq\_F4 , Float32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Select\_F4 , Float32x4\_1Int32x4\_1Float32x4\_2 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_And\_F4 , Float32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Or\_F4 , Float32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Xor\_F4 , Float32x4\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Not\_F4 , Float32x4\_2 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_LdSignMask\_F4 , Int1Float32x4\_1 , None , OpCanCSE , 0)

MACRO\_SIMD\_ASMJS\_ONLY\_WMS ( Simd128\_Ld\_F4 , Float32x4\_2 , None , None )

MACRO\_SIMD\_ASMJS\_ONLY\_WMS ( Simd128\_LdSlot\_F4 , ElementSlot , None , None )

MACRO\_SIMD\_ASMJS\_ONLY\_WMS ( Simd128\_StSlot\_F4 , ElementSlot , None , None )

MACRO\_SIMD\_ASMJS\_ONLY\_WMS ( Simd128\_Return\_F4 , Float32x4\_2 , None , None )

MACRO\_SIMD\_ASMJS\_ONLY\_WMS ( Simd128\_I\_ArgOut\_F4 , Reg1Float32x4\_1 , None , None )

MACRO\_SIMD\_ASMJS\_ONLY\_WMS ( Simd128\_I\_Conv\_VTF4 , Float32x4\_2 , None , None )

// Float64x2 , 0

MACRO\_SIMD\_WMS ( Simd128\_DoublesToD2 , Float64x2\_1Double2 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Splat\_D2 , Float64x2\_1Double1 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_FromFloat32x4\_D2 , Float64x2\_1Float32x4\_1 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_FromFloat32x4Bits\_D2 , Float64x2\_1Float32x4\_1 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_FromInt32x4\_D2 , Float64x2\_1Int32x4\_1 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_FromInt32x4Bits\_D2 , Float64x2\_1Int32x4\_1 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Abs\_D2 , Float64x2\_2 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Neg\_D2 , Float64x2\_2 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Add\_D2 , Float64x2\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Sub\_D2 , Float64x2\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Mul\_D2 , Float64x2\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Div\_D2 , Float64x2\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Clamp\_D2 , Float64x2\_4 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Min\_D2 , Float64x2\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Max\_D2 , Float64x2\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Rcp\_D2 , Float64x2\_2 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_RcpSqrt\_D2 , Float64x2\_2 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Sqrt\_D2 , Float64x2\_2 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Lt\_D2 , Float64x2\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Select\_D2 , Float64x2\_1Int32x4\_1Float64x2\_2 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_LdSignMask\_D2 , Int1Float64x2\_1 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_LtEq\_D2 , Float64x2\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Eq\_D2 , Float64x2\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Neq\_D2 , Float64x2\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_Gt\_D2 , Float64x2\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_WMS ( Simd128\_GtEq\_D2 , Float64x2\_3 , None , OpCanCSE , 0)

MACRO\_SIMD\_ASMJS\_ONLY\_WMS ( Simd128\_Return\_D2 , Float64x2\_2 , None , None )

MACRO\_SIMD\_ASMJS\_ONLY\_WMS ( Simd128\_I\_ArgOut\_D2 , Reg1Float64x2\_1 , None , None )

MACRO\_SIMD\_ASMJS\_ONLY\_WMS ( Simd128\_I\_Conv\_VTD2 , Float64x2\_2 , None , None )

MACRO\_SIMD\_BACKEND\_ONLY ( Simd128\_LdC , Empty , None , OpCanCSE ) // Load Simd128 const stack slot

MACRO\_SIMD ( Simd128\_End , Empty , None , None ) // Just a marker to indicate SIMD opcodes region

// Extended opcodes. Running out of 1-byte opcode space. Add new opcodes here.

MACRO\_SIMD\_EXTEND ( Simd128\_Start\_Extend , Empty , None , None , 0) // Just a marker to indicate SIMD opcodes region

MACRO\_SIMD\_ASMJS\_ONLY\_EXTEND\_WMS ( Simd128\_Ld\_D2 , Float64x2\_2 , None , None )

MACRO\_SIMD\_ASMJS\_ONLY\_EXTEND\_WMS ( Simd128\_LdSlot\_D2 , ElementSlot , None , None )

MACRO\_SIMD\_ASMJS\_ONLY\_EXTEND\_WMS ( Simd128\_StSlot\_D2 , ElementSlot , None , None )

MACRO\_SIMD\_EXTEND\_WMS ( Simd128\_Swizzle\_D2 , Float64x2\_2Int2 , None , OpCanCSE , 0)

MACRO\_SIMD\_EXTEND\_WMS ( Simd128\_Shuffle\_D2 , Float64x2\_3Int2 , None , OpCanCSE , 0)

MACRO\_SIMD\_EXTEND\_WMS ( Simd128\_LdArr\_I4 , AsmSimdTypedArr , None , OpCanCSE , 0)

MACRO\_SIMD\_EXTEND\_WMS ( Simd128\_LdArrConst\_I4 , AsmSimdTypedArr , None , OpCanCSE , 0)

MACRO\_SIMD\_EXTEND\_WMS ( Simd128\_LdArr\_F4 , AsmSimdTypedArr , None , OpCanCSE , 0)

MACRO\_SIMD\_EXTEND\_WMS ( Simd128\_LdArrConst\_F4 , AsmSimdTypedArr , None , OpCanCSE , 0)

MACRO\_SIMD\_EXTEND\_WMS ( Simd128\_LdArr\_D2 , AsmSimdTypedArr , None , OpCanCSE , 0)

MACRO\_SIMD\_EXTEND\_WMS ( Simd128\_LdArrConst\_D2 , AsmSimdTypedArr , None , OpCanCSE , 0)

MACRO\_SIMD\_EXTEND\_WMS ( Simd128\_StArr\_I4 , AsmSimdTypedArr , None , OpCanCSE , 0)

MACRO\_SIMD\_EXTEND\_WMS ( Simd128\_StArrConst\_I4 , AsmSimdTypedArr , None , OpCanCSE , 0)

MACRO\_SIMD\_EXTEND\_WMS ( Simd128\_StArr\_F4 , AsmSimdTypedArr , None , OpCanCSE , 0)

MACRO\_SIMD\_EXTEND\_WMS ( Simd128\_StArrConst\_F4 , AsmSimdTypedArr , None , OpCanCSE , 0)

MACRO\_SIMD\_EXTEND\_WMS ( Simd128\_StArr\_D2 , AsmSimdTypedArr , None , OpCanCSE , 0)

MACRO\_SIMD\_EXTEND\_WMS ( Simd128\_StArrConst\_D2 , AsmSimdTypedArr , None , OpCanCSE , 0)

MACRO\_SIMD\_EXTEND\_WMS ( Simd128\_ExtractLane\_I4 , Int1Int32x4\_1Int1 , None , OpCanCSE , 0)

MACRO\_SIMD\_EXTEND\_WMS ( Simd128\_ReplaceLane\_I4 , Int32x4\_2Int2 , None , OpCanCSE , 0)

MACRO\_SIMD\_EXTEND\_WMS ( Simd128\_ExtractLane\_F4 , Float1Float32x4\_1Int1 , None , OpCanCSE , 0)

MACRO\_SIMD\_EXTEND\_WMS ( Simd128\_ReplaceLane\_F4 , Float32x4\_2Int1Float1 , None , OpCanCSE , 0)

MACRO\_SIMD\_EXTEND ( Simd128\_End\_Extend , Empty , None , None , 0) // Just a marker to indicate SIMD opcodes region

#undef MACRO\_SIMD

#undef MACRO\_SIMD\_WMS

#undef MACRO\_SIMD\_ASMJS\_ONLY\_WMS

#undef MACRO\_SIMD\_BACKEND\_ONLY

#undef MACRO\_SIMD\_EXTEND

#undef MACRO\_SIMD\_EXTEND\_WMS

#undef MACRO\_SIMD\_ASMJS\_ONLY\_EXTEND\_WMS

#undef MACRO\_SIMD\_BACKEND\_ONLY\_EXTEND

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

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

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

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

#include "RuntimeByteCodePch.h"

namespace Js

{

bool OpCodeUtil::IsPrefixOpcode(OpCode op)

{

return op <= OpCode::ExtendedLargeLayoutPrefix && op != OpCode::EndOfBlock;

}

bool OpCodeUtil::IsSmallEncodedOpcode(OpCode op)

{

return op <= Js::OpCode::MaxByteSizedOpcodes;

}

uint OpCodeUtil::EncodedSize(OpCode op, LayoutSize layoutSize)

{

return (layoutSize == SmallLayout && IsSmallEncodedOpcode(op)) ? sizeof(BYTE) : sizeof(OpCode);

}

void OpCodeUtil::ConvertOpToNonProfiled(OpCode& op)

{

if (IsProfiledCallOp(op) || IsProfiledCallOpWithICIndex(op))

{

op = ConvertProfiledCallOpToNonProfiled(op);

}

else if (IsProfiledReturnTypeCallOp(op))

{

op = ConvertProfiledReturnTypeCallOpToNonProfiled(op);

}

else

{

ConvertNonCallOpToNonProfiled(op);

}

}

void OpCodeUtil::ConvertNonCallOpToProfiled(OpCode& op)

{

Assert(OpCodeAttr::HasProfiledOp(op));

op += 1;

Assert(OpCodeAttr::IsProfiledOp(op));

}

void OpCodeUtil::ConvertNonCallOpToProfiledWithICIndex(OpCode& op)

{

Assert(OpCodeAttr::HasProfiledOp(op) && OpCodeAttr::HasProfiledOpWithICIndex(op));

op += 2;

Assert(OpCodeAttr::IsProfiledOpWithICIndex(op));

}

void OpCodeUtil::ConvertNonCallOpToNonProfiled(OpCode& op)

{

if (OpCodeAttr::IsProfiledOp(op))

{

op -= 1;

Assert(OpCodeAttr::HasProfiledOp(op));

}

else if (OpCodeAttr::IsProfiledOpWithICIndex(op))

{

op -= 2;

Assert(OpCodeAttr::HasProfiledOpWithICIndex(op));

}

else

{

Assert(false);

}

}

bool OpCodeUtil::IsCallOp(OpCode op)

{

return op >= Js::OpCode::CallI && op <= Js::OpCode::CallIExtendedFlags;

}

bool OpCodeUtil::IsProfiledCallOp(OpCode op)

{

return op >= Js::OpCode::ProfiledCallI && op <= Js::OpCode::ProfiledCallIExtendedFlags;

}

bool OpCodeUtil::IsProfiledCallOpWithICIndex(OpCode op)

{

return op >= Js::OpCode::ProfiledCallIWithICIndex && op <= Js::OpCode::ProfiledCallIExtendedFlagsWithICIndex;

}

bool OpCodeUtil::IsProfiledReturnTypeCallOp(OpCode op)

{

return op >= Js::OpCode::ProfiledReturnTypeCallI && op <= Js::OpCode::ProfiledReturnTypeCallIExtendedFlags;

}

#if DBG

OpCode OpCodeUtil::DebugConvertProfiledCallToNonProfiled(OpCode op)

{

switch (op)

{

case Js::OpCode::ProfiledCallI:

case Js::OpCode::ProfiledCallIWithICIndex:

return Js::OpCode::CallI;

case Js::OpCode::ProfiledCallIFlags:

case Js::OpCode::ProfiledCallIFlagsWithICIndex:

return Js::OpCode::CallIFlags;

case Js::OpCode::ProfiledCallIExtendedFlags:

case Js::OpCode::ProfiledCallIExtendedFlagsWithICIndex:

return Js::OpCode::CallIExtendedFlags;

case Js::OpCode::ProfiledCallIExtended:

case Js::OpCode::ProfiledCallIExtendedWithICIndex:

return Js::OpCode::CallIExtended;

default:

Assert(false);

};

return Js::OpCode::Nop;

}

OpCode OpCodeUtil::DebugConvertProfiledReturnTypeCallToNonProfiled(OpCode op)

{

switch (op)

{

case Js::OpCode::ProfiledReturnTypeCallI:

return Js::OpCode::CallI;

case Js::OpCode::ProfiledReturnTypeCallIFlags:

return Js::OpCode::CallIFlags;

case Js::OpCode::ProfiledReturnTypeCallIExtendedFlags:

return Js::OpCode::CallIExtendedFlags;

case Js::OpCode::ProfiledReturnTypeCallIExtended:

return Js::OpCode::CallIExtended;

default:

Assert(false);

};

return Js::OpCode::Nop;

}

#endif

OpCode OpCodeUtil::ConvertProfiledCallOpToNonProfiled(OpCode op)

{

OpCode newOpcode;

if (IsProfiledCallOp(op))

{

newOpcode = (OpCode)(op - Js::OpCode::ProfiledCallI + Js::OpCode::CallI);

}

else if (IsProfiledCallOpWithICIndex(op))

{

newOpcode = (OpCode)(op - Js::OpCode::ProfiledCallIWithICIndex + Js::OpCode::CallI);

}

else

{

Assert(false);

\_\_assume(false);

}

Assert(DebugConvertProfiledCallToNonProfiled(op) == newOpcode);

return newOpcode;

}

OpCode OpCodeUtil::ConvertProfiledReturnTypeCallOpToNonProfiled(OpCode op)

{

OpCode newOpcode;

if (IsProfiledReturnTypeCallOp(op))

{

newOpcode = (OpCode)(op - Js::OpCode::ProfiledReturnTypeCallI + Js::OpCode::CallI);

}

else

{

Assert(false);

\_\_assume(false);

}

Assert(DebugConvertProfiledReturnTypeCallToNonProfiled(op) == newOpcode);

return newOpcode;

}

OpCode OpCodeUtil::ConvertCallOpToProfiled(OpCode op, bool withICIndex)

{

return (!withICIndex) ?

(OpCode)(op - OpCode::CallI + OpCode::ProfiledCallI) :

(OpCode)(op - OpCode::CallI + OpCode::ProfiledCallIWithICIndex);

}

OpCode OpCodeUtil::ConvertCallOpToProfiledReturnType(OpCode op)

{

return (OpCode)(op - OpCode::CallI + OpCode::ProfiledReturnTypeCallI);

}

CompileAssert(((int)Js::OpCode::CallIExtendedFlags - (int)Js::OpCode::CallI) == ((int)Js::OpCode::ProfiledCallIExtendedFlags - (int)Js::OpCode::ProfiledCallI));

CompileAssert(((int)Js::OpCode::CallIExtendedFlags - (int)Js::OpCode::CallI) == ((int)Js::OpCode::ProfiledReturnTypeCallIExtendedFlags - (int)Js::OpCode::ProfiledReturnTypeCallI));

CompileAssert(((int)Js::OpCode::CallIExtendedFlags - (int)Js::OpCode::CallI) == ((int)Js::OpCode::ProfiledCallIExtendedFlagsWithICIndex - (int)Js::OpCode::ProfiledCallIWithICIndex));

// Only include the opcode name on debug and test build

#if DBG\_DUMP || ENABLE\_DEBUG\_CONFIG\_OPTIONS

wchar\_t const \* const OpCodeUtil::OpCodeNames[] =

{

#define DEF\_OP(x, y, ...) L"" STRINGIZEW(x) L"",

#include "OpCodeList.h"

#undef DEF\_OP

};

wchar\_t const \* const OpCodeUtil::ExtendedOpCodeNames[] =

{

#define DEF\_OP(x, y, ...) L"" STRINGIZEW(x) L"",

#include "ExtendedOpCodeList.h"

#undef DEF\_OP

};

wchar\_t const \* const OpCodeUtil::BackendOpCodeNames[] =

{

#define DEF\_OP(x, y, ...) L"" STRINGIZEW(x) L"",

#include "BackendOpCodeList.h"

#undef DEF\_OP

};

wchar\_t const \* OpCodeUtil::GetOpCodeName(OpCode op)

{

if (op <= Js::OpCode::MaxByteSizedOpcodes)

{

Assert((uint)op < \_countof(OpCodeNames));

\_\_analysis\_assume((uint)op < \_countof(OpCodeNames));

return OpCodeNames[(uint)op];

}

else if (op < Js::OpCode::ByteCodeLast)

{

uint opIndex = op - (Js::OpCode::MaxByteSizedOpcodes + 1);

Assert(opIndex < \_countof(ExtendedOpCodeNames));

\_\_analysis\_assume(opIndex < \_countof(ExtendedOpCodeNames));

return ExtendedOpCodeNames[opIndex];

}

uint opIndex = op - (Js::OpCode::ByteCodeLast + 1);

Assert(opIndex < \_countof(BackendOpCodeNames));

\_\_analysis\_assume(opIndex < \_countof(BackendOpCodeNames));

return BackendOpCodeNames[opIndex];

}

#else

wchar const \* OpCodeUtil::GetOpCodeName(OpCode op)

{

return L"<NotAvail>";

}

#endif

OpLayoutType const OpCodeUtil::OpCodeLayouts[] =

{

#define DEF\_OP(x, y, ...) OpLayoutType::y,

#include "OpCodeList.h"

};

OpLayoutType const OpCodeUtil::ExtendedOpCodeLayouts[] =

{

#define DEF\_OP(x, y, ...) OpLayoutType::y,

#include "ExtendedOpCodeList.h"

};

OpLayoutType const OpCodeUtil::BackendOpCodeLayouts[] =

{

#define DEF\_OP(x, y, ...) OpLayoutType::y,

#include "BackendOpCodeList.h"

};

OpLayoutType OpCodeUtil::GetOpCodeLayout(OpCode op)

{

if ((uint)op <= (uint)Js::OpCode::MaxByteSizedOpcodes)

{

AnalysisAssert((uint)op < \_countof(OpCodeLayouts));

return OpCodeLayouts[(uint)op];

}

else if (op < Js::OpCode::ByteCodeLast)

{

uint opIndex = op - (Js::OpCode::MaxByteSizedOpcodes + 1);

AnalysisAssert(opIndex < \_countof(ExtendedOpCodeLayouts));

return ExtendedOpCodeLayouts[opIndex];

}

uint opIndex = op - (Js::OpCode::ByteCodeLast + 1);

AnalysisAssert(opIndex < \_countof(BackendOpCodeLayouts));

return BackendOpCodeLayouts[opIndex];

}

bool OpCodeUtil::IsValidByteCodeOpcode(OpCode op)

{

CompileAssert((int)Js::OpCode::MaxByteSizedOpcodes + 1 + \_countof(OpCodeUtil::ExtendedOpCodeLayouts) == (int)Js::OpCode::ByteCodeLast);

return (uint)op < \_countof(OpCodeLayouts)

|| (op > Js::OpCode::MaxByteSizedOpcodes && op < Js::OpCode::ByteCodeLast);

}

bool OpCodeUtil::IsValidOpcode(OpCode op)

{

return IsValidByteCodeOpcode(op)

|| (op > Js::OpCode::ByteCodeLast && op < Js::OpCode::Count);

}

};

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

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

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

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

#pragma once

namespace Js

{

class OpCodeUtil

{

public:

static wchar\_t const \* GetOpCodeName(OpCode op);

static bool IsCallOp(OpCode op);

static bool IsProfiledCallOp(OpCode op);

static bool IsProfiledCallOpWithICIndex(OpCode op);

static bool IsProfiledReturnTypeCallOp(OpCode op);

// OpCode conversion functions

static void ConvertOpToNonProfiled(OpCode& op);

static void ConvertNonCallOpToProfiled(OpCode& op);

static void ConvertNonCallOpToProfiledWithICIndex(OpCode& op);

static void ConvertNonCallOpToNonProfiled(OpCode& op);

static void ConvertNonCallOpToNonProfiledWithICIndex(OpCode& op);

static OpCode ConvertProfiledCallOpToNonProfiled(OpCode op);

static OpCode ConvertProfiledReturnTypeCallOpToNonProfiled(OpCode op);

static OpCode ConvertCallOpToProfiled(OpCode op, bool withICIndex = false);

static OpCode ConvertCallOpToProfiledReturnType(OpCode op);

static bool IsValidByteCodeOpcode(OpCode op);

static bool IsValidOpcode(OpCode op);

static bool IsPrefixOpcode(OpCode op);

static bool IsSmallEncodedOpcode(OpCode op);

static uint EncodedSize(OpCode op, LayoutSize layoutSize);

static OpLayoutType GetOpCodeLayout(OpCode op);

private:

#if DBG\_DUMP || ENABLE\_DEBUG\_CONFIG\_OPTIONS

static wchar\_t const \* const OpCodeNames[(int)Js::OpCode::MaxByteSizedOpcodes + 1];

static wchar\_t const \* const ExtendedOpCodeNames[];

static wchar\_t const \* const BackendOpCodeNames[];

#endif

static OpLayoutType const OpCodeLayouts[];

static OpLayoutType const ExtendedOpCodeLayouts[];

static OpLayoutType const BackendOpCodeLayouts[];

#if DBG

static OpCode DebugConvertProfiledCallToNonProfiled(OpCode op);

static OpCode DebugConvertProfiledReturnTypeCallToNonProfiled(OpCode op);

#endif

};

};

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

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

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

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

#include "RuntimeByteCodePch.h"

#ifndef TEMP\_DISABLE\_ASMJS

namespace Js

{

#if DBG\_DUMP || ENABLE\_DEBUG\_CONFIG\_OPTIONS

wchar\_t const \* const OpCodeUtilAsmJs::OpCodeAsmJsNames[] =

{

#define DEF\_OP(x, y, ...) L"" STRINGIZEW(x) L"",

#include "OpCodeListAsmJs.h"

#undef DEF\_OP

};

wchar\_t const \* const OpCodeUtilAsmJs::ExtendedOpCodeAsmJsNames[] =

{

#define DEF\_OP(x, y, ...) L"" STRINGIZEW(x) L"",

#include "ExtendedOpCodeListAsmJs.h"

#undef DEF\_OP

};

wchar\_t const \* OpCodeUtilAsmJs::GetOpCodeName(OpCodeAsmJs op)

{

if (op <= Js::OpCodeAsmJs::MaxByteSizedOpcodes)

{

Assert(op < \_countof(OpCodeAsmJsNames));

\_\_analysis\_assume(op < \_countof(OpCodeAsmJsNames));

return OpCodeAsmJsNames[(int)op];

}

else if (op < Js::OpCodeAsmJs::ByteCodeLast)

{

uint opIndex = op - (Js::OpCodeAsmJs::MaxByteSizedOpcodes + 1);

Assert(opIndex < \_countof(ExtendedOpCodeAsmJsNames));

\_\_analysis\_assume(opIndex < \_countof(ExtendedOpCodeAsmJsNames));

return ExtendedOpCodeAsmJsNames[opIndex];

}

return L"<NotAvail>";

}

#else

wchar const \* OpCodeUtilAsmJs::GetOpCodeName(OpCodeAsmJs op)

{

return L"<NotAvail>";

}

#endif

OpLayoutTypeAsmJs const OpCodeUtilAsmJs::OpCodeAsmJsLayouts[] =

{

#define DEF\_OP(x, y, ...) OpLayoutTypeAsmJs::y,

#include "OpCodeListAsmJs.h"

};

OpLayoutTypeAsmJs const OpCodeUtilAsmJs::ExtendedOpCodeAsmJsLayouts[] =

{

#define DEF\_OP(x, y, ...) OpLayoutTypeAsmJs::y,

#include "ExtendedOpCodeListAsmJs.h"

};

OpLayoutTypeAsmJs OpCodeUtilAsmJs::GetOpCodeLayout(OpCodeAsmJs op)

{

if ((uint)op <= (uint)Js::OpCodeAsmJs::MaxByteSizedOpcodes)

{

Assert(op < \_countof(OpCodeAsmJsLayouts));

\_\_analysis\_assume(op < \_countof(OpCodeAsmJsLayouts));

return OpCodeAsmJsLayouts[(uint)op];

}

uint opIndex = op - (Js::OpCodeAsmJs::MaxByteSizedOpcodes + 1);

Assert(opIndex < \_countof(ExtendedOpCodeAsmJsLayouts));

\_\_analysis\_assume(opIndex < \_countof(ExtendedOpCodeAsmJsLayouts));

return ExtendedOpCodeAsmJsLayouts[opIndex];

}

bool OpCodeUtilAsmJs::IsValidByteCodeOpcode(OpCodeAsmJs op)

{

CompileAssert((int)Js::OpCodeAsmJs::MaxByteSizedOpcodes + 1 + \_countof(OpCodeUtilAsmJs::ExtendedOpCodeAsmJsLayouts) == (int)Js::OpCodeAsmJs::ByteCodeLast);

return op < \_countof(OpCodeAsmJsLayouts)

|| (op > Js::OpCodeAsmJs::MaxByteSizedOpcodes && op < Js::OpCodeAsmJs::ByteCodeLast);

}

bool OpCodeUtilAsmJs::IsValidOpcode(OpCodeAsmJs op)

{

return IsValidByteCodeOpcode(op)

|| (op > Js::OpCodeAsmJs::ByteCodeLast && op < Js::OpCodeAsmJs::Count);

}

};

#endif

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

// 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

#ifndef TEMP\_DISABLE\_ASMJS

namespace Js

{

class OpCodeUtilAsmJs

{

public:

static wchar\_t const \* GetOpCodeName(OpCodeAsmJs op);

static OpLayoutTypeAsmJs GetOpCodeLayout(OpCodeAsmJs op);

static bool IsValidByteCodeOpcode(OpCodeAsmJs op);

static bool IsValidOpcode(OpCodeAsmJs op);

private:

#if DBG\_DUMP || ENABLE\_DEBUG\_CONFIG\_OPTIONS

static wchar\_t const \* const OpCodeAsmJsNames[(int)Js::OpCodeAsmJs::MaxByteSizedOpcodes + 1];

static wchar\_t const \* const ExtendedOpCodeAsmJsNames[];

#endif

static OpLayoutTypeAsmJs const OpCodeAsmJsLayouts[];

static OpLayoutTypeAsmJs const ExtendedOpCodeAsmJsLayouts[];

};

};

#endif

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

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

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

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

#pragma once

namespace Js {

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

///

/// enum OpCode

///

/// OpCode defines the set of p-code instructions available for byte-code.

///

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

enum class OpCode : ushort

{

#define DEF\_OP(x, y, ...) x,

#include "OpCodeList.h"

MaxByteSizedOpcodes = 255,

#include "ExtendedOpCodeList.h"

ByteCodeLast,

#if ENABLE\_NATIVE\_CODEGEN

#include "BackEndOpCodeList.h"

#endif

#undef DEF\_OP

Count // Number of operations

};

inline OpCode operator+(OpCode o1, OpCode o2) { return (OpCode)((uint)o1 + (uint)o2); }

inline uint operator+(OpCode o1, uint i) { return ((uint)o1 + i); }

inline uint operator+(uint i, OpCode &o2) { return (i + (uint)o2); }

inline OpCode operator++(OpCode &o) { return o = (OpCode)(o + 1U); }

inline OpCode operator++(OpCode &o, int) { OpCode prev\_o = o; o = (OpCode)(o + 1U); return prev\_o; }

inline OpCode operator-(OpCode o1, OpCode o2) { return (OpCode)((uint)o1 - (uint)o2); }

inline uint operator-(OpCode o1, uint i) { return ((uint)o1 - i); }

inline uint operator-(uint i, OpCode &o2) { return (i - (uint)o2); }

inline OpCode operator--(OpCode &o) { return o = (OpCode)(o - 1U); }

inline OpCode operator--(OpCode &o, int) { return o = (OpCode)(o - 1U); }

inline uint operator<<(OpCode o1, uint i) { return ((uint)o1 << i); }

inline OpCode& operator+=(OpCode &o, uint i) { return (o = (OpCode)(o + i)); }

inline OpCode& operator-=(OpCode &o, uint i) { return (o = (OpCode)(o - i)); }

inline bool operator==(OpCode &o, uint i) { return ((uint)(o) == i); }

inline bool operator==(uint i, OpCode &o) { return (i == (uint)(o)); }

inline bool operator!=(OpCode &o, uint i) { return ((uint)(o) != i); }

inline bool operator!=(uint i, OpCode &o) { return (i != (uint)(o)); }

inline bool operator<(OpCode &o, uint i) { return ((uint)(o) < i); }

inline bool operator<(uint i, OpCode &o) { return (i < (uint)(o)); }

inline bool operator>(OpCode &o, uint i) { return ((uint)(o) > i); }

inline bool operator>(uint i, OpCode &o) { return (i > (uint)(o)); }

#if ENABLE\_NATIVE\_CODEGEN

inline bool IsSimd128Opcode(OpCode o) { return (o > Js::OpCode::Simd128\_Start && o < Js::OpCode::Simd128\_End) || (o > Js::OpCode::Simd128\_Start\_Extend && o < Js::OpCode::Simd128\_End\_Extend); }

inline uint Simd128OpcodeCount() { return (uint)(Js::OpCode::Simd128\_End - Js::OpCode::Simd128\_Start) + 1 + (uint)(Js::OpCode::Simd128\_End\_Extend - Js::OpCode::Simd128\_Start\_Extend) + 1; }

#endif

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

///

/// enum OpLayoutType

///

/// OpLayoutType defines a set of layouts available for OpCodes. These layouts

/// correspond to "OpLayout" structs defined below, such as "OpLayoutReg1".

///

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

BEGIN\_ENUM\_UINT(OpLayoutType)

// This define only one enum for each layout type, but not for each layout variant

#define LAYOUT\_TYPE(x) x,

#define LAYOUT\_TYPE\_WMS LAYOUT\_TYPE

#include "LayoutTypes.h"

Count,

END\_ENUM\_UINT()

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

///

/// struct OpLayoutXYZ

///

/// OpLayoutXYZ structs define the standard patterns used to layout each

/// OpCode's arguments.

///

/// Set up packing:

/// - Since we are reading / writing from a byte-stream, and we want everything

/// to be tightly aligned with no unintended spaces, change to 'byte'

/// packing.

/// - On processors with alignment requirements, this will automatically

/// generate read / write code to handle 'unaligned' access.

///

/// - 3/9/10: Changing the layouts to make all fields well-aligned. This involves

/// reducing the RegSlot from 4 bytes to 2, increasing the OpCode from 1 byte to 2,

/// reordering fields, and adding pads where appropriate. Note that we assume all

/// layouts are preceded by a 2-byte op. The orderings and padding will need to

/// be revisited for Win64.

///

/// - 5/2: X86-64 alignment: Changing code to expect all opcode layout structs

/// to be aligned on 4 byte boundaries (on I386 and x86-64). This aligns all

/// members on their natural boundaries except for Call and Regex which require

/// padding before the pointers on x86-64.

///

/// - 09/12: Adding one-byte RegSlot based OpLayout. These remove all the paddings to be

/// able to compress the size. Also remove paddings for non ARM build

///

/// - 08/22/2011: Removed paddings for ARM & x64 as well as both support unaligned access

/// There is still paddings to make sure every opcode starts at 2 byte boundary to avoid

/// pathological cases.

/// Note: RegSlot is changed to 4 byte instead of 1 byte in this change.

///

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

#pragma pack(push, 1)

template <typename SizePolicy>

struct OpLayoutT\_Reg1 // R0 <- op

{

typename SizePolicy::RegSlotType R0;

};

template <typename SizePolicy>

struct OpLayoutT\_Reg1Unsigned1 // R0 <- op

{

typename SizePolicy::RegSlotType R0;

typename SizePolicy::UnsignedType C1;

};

template <typename SizePolicy>

struct OpLayoutT\_Reg2 // R0 <- op R1

{

typename SizePolicy::RegSlotType R0;

typename SizePolicy::RegSlotType R1;

};

template <typename SizePolicy>

struct OpLayoutT\_Reg2WithICIndex : public OpLayoutT\_Reg2<SizePolicy>

{

InlineCacheIndex inlineCacheIndex;

};

template <typename SizePolicy>

struct OpLayoutT\_Reg3 // R0 <- R1 op R2 -- or -- R0 op R1 <- R2

{

typename SizePolicy::RegSlotType R0;

typename SizePolicy::RegSlotType R1;

typename SizePolicy::RegSlotType R2;

};

template <typename SizePolicy>

struct OpLayoutT\_Reg3C // R0 <- R1 op R2 with space for FastPath

{

typename SizePolicy::RegSlotType R0;

typename SizePolicy::RegSlotType R1;

typename SizePolicy::RegSlotType R2;

typename SizePolicy::CacheIdType inlineCacheIndex;

};

template <typename SizePolicy>

struct OpLayoutT\_Reg2B1

{

typename SizePolicy::RegSlotType R0;

typename SizePolicy::RegSlotType R1;

byte B2;

};

template <typename SizePolicy>

struct OpLayoutT\_Reg3B1

{

typename SizePolicy::RegSlotType R0;

typename SizePolicy::RegSlotType R1;

typename SizePolicy::RegSlotType R2;

byte B3;

};

template <typename SizePolicy>

struct OpLayoutT\_Reg4 // R0 <- R1 op R2 op R3

{

typename SizePolicy::RegSlotType R0;

typename SizePolicy::RegSlotType R1;

typename SizePolicy::RegSlotType R2;

typename SizePolicy::RegSlotType R3;

};

template <typename SizePolicy>

struct OpLayoutT\_Reg5 // R0 <- R1 op R2 op R3 op R4

{

typename SizePolicy::RegSlotType R0;

typename SizePolicy::RegSlotType R1;

typename SizePolicy::RegSlotType R2;

typename SizePolicy::RegSlotType R3;

typename SizePolicy::RegSlotType R4;

};

template <typename SizePolicy>

struct OpLayoutT\_ArgNoSrc // OutArg

{

typename SizePolicy::ArgSlotType Arg;

};

template <typename SizePolicy>

struct OpLayoutT\_Arg // OutArg <- Reg -- or -- Reg <- InArg

{

typename SizePolicy::ArgSlotType Arg;

typename SizePolicy::RegSlotType Reg;

};

struct OpLayoutBr // goto Offset

{

JumpOffset RelativeJumpOffset;

};

struct OpLayoutBrS // if (op val) goto Offset

{

JumpOffset RelativeJumpOffset;

byte val;

};

template <typename SizePolicy>

struct OpLayoutT\_BrReg1 // if (op R1) goto Offset

{

JumpOffset RelativeJumpOffset;

typename SizePolicy::RegSlotType R1;

};

template <typename SizePolicy>

struct OpLayoutT\_BrReg2 // if (R1 op R2) goto Offset

{

JumpOffset RelativeJumpOffset;

typename SizePolicy::RegSlotType R1;

typename SizePolicy::RegSlotType R2;

};

struct OpLayoutBrProperty // if (R1.id) goto Offset

{

JumpOffset RelativeJumpOffset;

RegSlot Instance;

PropertyIdIndexType PropertyIdIndex;

};

struct OpLayoutBrLocalProperty // if (id) goto Offset

{

JumpOffset RelativeJumpOffset;

PropertyIdIndexType PropertyIdIndex;

};

struct OpLayoutBrEnvProperty // if ([1].id) goto Offset

{

JumpOffset RelativeJumpOffset;

PropertyIdIndexType PropertyIdIndex;

int32 SlotIndex;

};

#ifdef BYTECODE\_BRANCH\_ISLAND

struct OpLayoutBrLong

{

LongJumpOffset RelativeJumpOffset;

};

#endif

struct OpLayoutStartCall

{

ArgSlot ArgCount;

};

enum CallIExtendedOptions : byte

{

CallIExtended\_None = 0,

CallIExtended\_SpreadArgs = 1 << 0 // This call has arguments that need to be spread

};

template <typename SizePolicy>

struct OpLayoutT\_CallI // Return = Function(ArgCount)

{

typename SizePolicy::ArgSlotType ArgCount;

typename SizePolicy::RegSlotSType Return;

typename SizePolicy::RegSlotType Function;

};

template <typename SizePolicy>

struct OpLayoutT\_CallIExtended : public OpLayoutT\_CallI<SizePolicy>

{

CallIExtendedOptions Options;

uint32 SpreadAuxOffset; // Valid with Options & CallIExtended\_SpreadArgs

};

template <typename SizePolicy>

struct OpLayoutT\_CallIFlags : public OpLayoutT\_CallI<SizePolicy>

{

CallFlags callFlags;

};

template <typename SizePolicy>

struct OpLayoutT\_CallIWithICIndex : public OpLayoutT\_CallI<SizePolicy>

{

InlineCacheIndex inlineCacheIndex;

};

template <typename SizePolicy>

struct OpLayoutT\_CallIFlagsWithICIndex : public OpLayoutT\_CallIWithICIndex<SizePolicy>

{

CallFlags callFlags;

};

template <typename SizePolicy>

struct OpLayoutT\_CallIExtendedFlags : public OpLayoutT\_CallIExtended<SizePolicy>

{

CallFlags callFlags;

};

template <typename SizePolicy>

struct OpLayoutT\_CallIExtendedWithICIndex : public OpLayoutT\_CallIExtended<SizePolicy>

{

InlineCacheIndex inlineCacheIndex;

};

template <typename SizePolicy>

struct OpLayoutT\_CallIExtendedFlagsWithICIndex : public OpLayoutT\_CallIExtendedWithICIndex<SizePolicy>

{

CallFlags callFlags;

};

template <typename SizePolicy>

struct OpLayoutT\_Class // class \_ extends Extends { Constructor(...) { ... } }

{

typename SizePolicy::RegSlotType Constructor;

typename SizePolicy::RegSlotSType Extends;

};

template <typename SizePolicy>

struct OpLayoutT\_ElementU // Instance.PropertyIndex = <some constant value>. e.g. undefined

{

typename SizePolicy::RegSlotType Instance;

typename SizePolicy::PropertyIdIndexType PropertyIdIndex;

};

template <typename SizePolicy>

struct OpLayoutT\_ElementScopedU // [env].PropertyIndex = <some constant value>. e.g. undefined

{

typename SizePolicy::PropertyIdIndexType PropertyIdIndex;

};

template <typename SizePolicy>

struct OpLayoutT\_ElementRootU // Root.PropertyIndex = <some constant value>. e.g. undefined

{

typename SizePolicy::PropertyIdIndexType PropertyIdIndex;

};

template <typename SizePolicy>

struct OpLayoutT\_ElementC // Value = Instance.PropertyIndex or Instance.PropertyIndex = Value

{

typename SizePolicy::RegSlotType Value;

typename SizePolicy::RegSlotType Instance;

typename SizePolicy::PropertyIdIndexType PropertyIdIndex;

};

template <typename SizePolicy>

struct OpLayoutT\_ElementScopedC // Value = [env].PropertyIndex or [env].PropertyIndex = Value

{

typename SizePolicy::RegSlotType Value;

typename SizePolicy::PropertyIdIndexType PropertyIdIndex;

};

template <typename SizePolicy>

struct OpLayoutT\_ElementSlot // Value = Instance[SlotIndex] or Instance[SlotIndex] = Value

{

int32 SlotIndex; // TODO: Make this one byte?

typename SizePolicy::RegSlotType Value;

typename SizePolicy::RegSlotType Instance;

};

template <typename SizePolicy>

struct OpLayoutT\_ElementSlotI1

{

int32 SlotIndex; // TODO: Make this one byte?

typename SizePolicy::RegSlotType Value;

};

template <typename SizePolicy>

struct OpLayoutT\_ElementSlotI2

{

int32 SlotIndex1; // TODO: Make this one byte?

int32 SlotIndex2; // TODO: Make this one byte?

typename SizePolicy::RegSlotType Value;

};

template <typename SizePolicy>

struct OpLayoutT\_ElementCP // As OpLayoutElementC, with space for a FastPath LoadPatch

{

typename SizePolicy::RegSlotType Value;

typename SizePolicy::RegSlotType Instance;

typename SizePolicy::CacheIdType inlineCacheIndex;

};

template <typename SizePolicy>

struct OpLayoutT\_ElementP // As OpLayoutElementCP, but with no base pointer

{

typename SizePolicy::RegSlotType Value;

typename SizePolicy::CacheIdType inlineCacheIndex;

};

template <typename SizePolicy>

struct OpLayoutT\_ElementPIndexed // As OpLayoutElementCP, but with scope index instead of base pointer

{

typename SizePolicy::RegSlotType Value;

typename SizePolicy::CacheIdType inlineCacheIndex;

typename SizePolicy::UnsignedType scopeIndex;

};

template <typename SizePolicy>

struct OpLayoutT\_ElementRootCP // Same as ElementCP, but for root object

{

RootCacheId inlineCacheIndex;

typename SizePolicy::RegSlotType Value;

};

template <typename SizePolicy>

struct OpLayoutT\_ElementScopedC2 // [implied base].PropertyIndex = Value, Instance2

{

typename SizePolicy::RegSlotType Value;

typename SizePolicy::RegSlotType Value2;

typename SizePolicy::PropertyIdIndexType PropertyIdIndex;

};

template <typename SizePolicy>

struct OpLayoutT\_ElementC2 // Instance.PropertyIndex = Value, Instance2

{

typename SizePolicy::RegSlotType Value;

typename SizePolicy::RegSlotType Instance;

typename SizePolicy::RegSlotType Value2;

typename SizePolicy::PropertyIdIndexType PropertyIdIndex;

};

template <typename SizePolicy>

struct OpLayoutT\_ElementI // Value = Instance[Element] or Instance[Element] = Value

{

typename SizePolicy::RegSlotType Value;

typename SizePolicy::RegSlotType Instance;

typename SizePolicy::RegSlotType Element;

};

template <typename SizePolicy>

struct OpLayoutT\_ElementUnsigned1 // Value = Instance[Element] or Instance[Element] = Value

{

typename SizePolicy::UnsignedType Element;

typename SizePolicy::RegSlotType Value;

typename SizePolicy::RegSlotType Instance;

};

struct OpLayoutW1

{

unsigned short C1;

};

struct OpLayoutReg1Int2 // R0 <- Var(C1, C2)

{

RegSlot R0;

int32 C1;

int32 C2;

};

template <typename SizePolicy>

struct OpLayoutT\_Reg2Int1 // R0 <- func(R1, C1)

{

int32 C1;

typename SizePolicy::RegSlotType R0;

typename SizePolicy::RegSlotType R1;

};

struct OpLayoutAuxNoReg

{

uint32 Offset;

int32 C1;

};

struct OpLayoutAuxiliary : public OpLayoutAuxNoReg // R0 <- Load(Offset, C1)

{

RegSlot R0;

};

struct OpLayoutReg2Aux : public OpLayoutAuxiliary // R0 <- Load(Offset, R1, C1)

{

RegSlot R1;

};

template <typename SizePolicy>

struct OpLayoutT\_Unsigned1

{

typename SizePolicy::UnsignedType C1;

};

// Dynamic profile layout wrapper

template <typename LayoutType>

struct OpLayoutDynamicProfile : public LayoutType

{

ProfileId profileId;

};

template <typename LayoutType>

struct OpLayoutDynamicProfile2 : public LayoutType

{

ProfileId profileId;

ProfileId profileId2;

};

// Generate the multi size layout type defs

#define LAYOUT\_TYPE\_WMS(layout) \

typedef OpLayoutT\_##layout<LargeLayoutSizePolicy> OpLayout##layout##\_Large; \

typedef OpLayoutT\_##layout<MediumLayoutSizePolicy> OpLayout##layout##\_Medium; \

typedef OpLayoutT\_##layout<SmallLayoutSizePolicy> OpLayout##layout##\_Small;

// Generate the profiled type defs

#define LAYOUT\_TYPE\_PROFILED(layout) \

typedef OpLayoutDynamicProfile<OpLayout##layout> OpLayoutProfiled##layout;

#define LAYOUT\_TYPE\_PROFILED2(layout) \

typedef OpLayoutDynamicProfile2<OpLayout##layout> OpLayoutProfiled2##layout;

#define LAYOUT\_TYPE\_PROFILED\_WMS(layout) \

LAYOUT\_TYPE\_WMS(layout) \

LAYOUT\_TYPE\_PROFILED(layout##\_Large) \

LAYOUT\_TYPE\_PROFILED(layout##\_Medium) \

LAYOUT\_TYPE\_PROFILED(layout##\_Small)

#define LAYOUT\_TYPE\_PROFILED2\_WMS(layout) \

LAYOUT\_TYPE\_PROFILED\_WMS(layout) \

LAYOUT\_TYPE\_PROFILED2(layout##\_Large) \

LAYOUT\_TYPE\_PROFILED2(layout##\_Medium) \

LAYOUT\_TYPE\_PROFILED2(layout##\_Small)

#include "LayoutTypes.h"

#pragma pack(pop)

// Generate structure to automatically map layout to its info

template <OpLayoutType::\_E layout> struct OpLayoutInfo;

#define LAYOUT\_TYPE(layout) \

CompileAssert(sizeof(OpLayout##layout) <= MaxLayoutSize); \

template <> struct OpLayoutInfo<OpLayoutType::layout> \

{ \

static const bool HasMultiSizeLayout = false; \

};

#define LAYOUT\_TYPE\_WMS(layout) \

CompileAssert(sizeof(OpLayout##layout##\_Large) <= MaxLayoutSize); \

template <> struct OpLayoutInfo<OpLayoutType::layout> \

{ \

static const bool HasMultiSizeLayout = true; \

};

#include "LayoutTypes.h"

// Generate structure to automatically map opcode to its info

// Also generate assert to make sure the layout and opcode use the same macro with and without multiple size layout

template <OpCode opcode> struct OpCodeInfo;

#define DEFINE\_OPCODEINFO(op, layout, extended) \

CompileAssert(!OpLayoutInfo<OpLayoutType::layout>::HasMultiSizeLayout); \

template <> struct OpCodeInfo<OpCode::op> \

{ \

static const OpLayoutType::\_E Layout = OpLayoutType::layout; \

static const bool HasMultiSizeLayout = false; \

static const bool IsExtendedOpcode = extended; \

typedef OpLayout##layout LayoutType; \

};

#define DEFINE\_OPCODEINFO\_WMS(op, layout, extended) \

CompileAssert(OpLayoutInfo<OpLayoutType::layout>::HasMultiSizeLayout); \

template <> struct OpCodeInfo<OpCode::op> \

{ \

static const OpLayoutType::\_E Layout = OpLayoutType::layout; \

static const bool HasMultiSizeLayout = true; \

static const bool IsExtendedOpcode = extended; \

typedef OpLayout##layout##\_Large LayoutType\_Large; \

typedef OpLayout##layout##\_Medium LayoutType\_Medium; \

typedef OpLayout##layout##\_Small LayoutType\_Small; \

};

#define MACRO(op, layout, ...) DEFINE\_OPCODEINFO(op, layout, false)

#define MACRO\_WMS(op, layout, ...) DEFINE\_OPCODEINFO\_WMS(op, layout, false)

#define MACRO\_EXTEND(op, layout, ...) DEFINE\_OPCODEINFO(op, layout, true)

#define MACRO\_EXTEND\_WMS(op, layout, ...) DEFINE\_OPCODEINFO\_WMS(op, layout, true)

#include "OpCodes.h"

#undef DEFINE\_OPCODEINFO

#undef DEFINE\_OPCODEINFO\_WMS

} // 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.

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

#pragma once

#ifndef TEMP\_DISABLE\_ASMJS

namespace Js

{

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

///

/// enum OpCodeAsmJs

///

/// OpCodeAsmJs defines the set of p-code instructions available for byte-code in Asm.Js.

///

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

enum class OpCodeAsmJs : ushort {

#define DEF\_OP(x, y, ...) x,

#include "OpCodeListAsmJs.h"

MaxByteSizedOpcodes = 255,

#include "ExtendedOpCodeListAsmJs.h"

ByteCodeLast,

#undef DEF\_OP

Count // Number of operations

};

inline OpCodeAsmJs operator+(OpCodeAsmJs o1, OpCodeAsmJs o2) { return (OpCodeAsmJs)((uint)o1 + (uint)o2); }

inline uint operator+(OpCodeAsmJs o1, uint i) { return ((uint)o1 + i); }

inline uint operator+(uint i, OpCodeAsmJs &o2) { return (i + (uint)o2); }

inline OpCodeAsmJs operator++(OpCodeAsmJs &o) { return o = (OpCodeAsmJs)(o + 1U); }

inline OpCodeAsmJs operator++(OpCodeAsmJs &o, int) { OpCodeAsmJs prev\_o = o; o = (OpCodeAsmJs)(o + 1U); return prev\_o; }

inline OpCodeAsmJs operator-(OpCodeAsmJs o1, OpCodeAsmJs o2) { return (OpCodeAsmJs)((uint)o1 - (uint)o2); }

inline uint operator-(OpCodeAsmJs o1, uint i) { return ((uint)o1 - i); }

inline uint operator-(uint i, OpCodeAsmJs &o2) { return (i - (uint)o2); }

inline OpCodeAsmJs operator--(OpCodeAsmJs &o) { return o = (OpCodeAsmJs)(o - 1U); }

inline OpCodeAsmJs operator--(OpCodeAsmJs &o, int) { return o = (OpCodeAsmJs)(o - 1U); }

inline uint operator<<(OpCodeAsmJs o1, uint i) { return ((uint)o1 << i); }

inline OpCodeAsmJs& operator+=(OpCodeAsmJs &o, uint i) { return (o = (OpCodeAsmJs)(o + i)); }

inline OpCodeAsmJs& operator-=(OpCodeAsmJs &o, uint i) { return (o = (OpCodeAsmJs)(o - i)); }

inline bool operator==(OpCodeAsmJs &o, uint i) { return ((uint)(o) == i); }

inline bool operator==(uint i, OpCodeAsmJs &o) { return (i == (uint)(o)); }

inline bool operator!=(OpCodeAsmJs &o, uint i) { return ((uint)(o) != i); }

inline bool operator!=(uint i, OpCodeAsmJs &o) { return (i != (uint)(o)); }

inline bool operator<(OpCodeAsmJs &o, uint i) { return ((uint)(o) < i); }

inline bool operator<(uint i, OpCodeAsmJs &o) { return (i < (uint)(o)); }

inline bool operator<=(OpCodeAsmJs &o, uint i) { return ((uint)(o) <= i); }

inline bool operator<=(uint i, OpCodeAsmJs &o) { return (i <= (uint)(o)); }

inline bool operator<=(OpCodeAsmJs o1, OpCode o2) { return ((OpCode)o1 <= (o2)); }

inline bool operator>(OpCodeAsmJs &o, uint i) { return ((uint)(o) > i); }

inline bool operator>(uint i, OpCodeAsmJs &o) { return (i > (uint)(o)); }

inline bool operator>=(OpCodeAsmJs &o, uint i) { return ((uint)(o) >= i); }

inline bool operator>=(uint i, OpCodeAsmJs &o) { return (i >= (uint)(o)); }

inline bool IsSimd128AsmJsOpcode(OpCodeAsmJs o)

{

return (o > Js::OpCodeAsmJs::Simd128\_Start && o < Js::OpCodeAsmJs::Simd128\_End) || (o > Js::OpCodeAsmJs::Simd128\_Start\_Extend && o < Js::OpCodeAsmJs::Simd128\_End\_Extend);

}

inline uint Simd128AsmJsOpcodeCount()

{

return (uint)(Js::OpCodeAsmJs::Simd128\_End - Js::OpCodeAsmJs::Simd128\_Start) + 1 + (uint)(Js::OpCodeAsmJs::Simd128\_End\_Extend - Js::OpCodeAsmJs::Simd128\_Start\_Extend) + 1;

}

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

///

/// enum OpLayoutTypeAsmJs

///

/// OpLayoutTypeAsmJs defines a set of layouts available for OpCodes. These layouts

/// correspond to "OpLayout" structs defined below, such as "OpLayoutReg1".

///

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

BEGIN\_ENUM\_UINT( OpLayoutTypeAsmJs )

// This define only one enum for each layout type, but not for each layout variant

#define LAYOUT\_TYPE(x) x,

#define LAYOUT\_TYPE\_WMS LAYOUT\_TYPE

#include "LayoutTypesAsmJs.h"

Count,

END\_ENUM\_UINT()

#pragma pack(push, 1)

/// Asm.js Layout

template <typename SizePolicy>

struct OpLayoutT\_AsmTypedArr

{

// force encode 4 bytes because it can be a value

uint32 SlotIndex;

typename SizePolicy::RegSlotType Value;

int8 ViewType;

};

template <typename SizePolicy>

struct OpLayoutT\_AsmCall

{

typename SizePolicy::ArgSlotType ArgCount;

typename SizePolicy::RegSlotSType Return;

typename SizePolicy::RegSlotType Function;

int8 ReturnType;

};

template <typename SizePolicy>

struct OpLayoutT\_AsmReg1

{

typename SizePolicy::RegSlotType R0;

};

template <typename SizePolicy>

struct OpLayoutT\_AsmReg2

{

typename SizePolicy::RegSlotType R0;

typename SizePolicy::RegSlotType R1;

};

template <typename SizePolicy>

struct OpLayoutT\_AsmReg3

{

typename SizePolicy::RegSlotType R0;

typename SizePolicy::RegSlotType R1;

typename SizePolicy::RegSlotType R2;

};

template <typename SizePolicy>

struct OpLayoutT\_AsmReg4

{

typename SizePolicy::RegSlotType R0;

typename SizePolicy::RegSlotType R1;

typename SizePolicy::RegSlotType R2;

typename SizePolicy::RegSlotType R3;

};

template <typename SizePolicy>

struct OpLayoutT\_AsmReg5

{

typename SizePolicy::RegSlotType R0;

typename SizePolicy::RegSlotType R1;

typename SizePolicy::RegSlotType R2;

typename SizePolicy::RegSlotType R3;

typename SizePolicy::RegSlotType R4;

};

template <typename SizePolicy>

struct OpLayoutT\_AsmReg6

{

typename SizePolicy::RegSlotType R0;

typename SizePolicy::RegSlotType R1;

typename SizePolicy::RegSlotType R2;

typename SizePolicy::RegSlotType R3;

typename SizePolicy::RegSlotType R4;

typename SizePolicy::RegSlotType R5;

};

template <typename SizePolicy>

struct OpLayoutT\_AsmReg7

{

typename SizePolicy::RegSlotType R0;

typename SizePolicy::RegSlotType R1;

typename SizePolicy::RegSlotType R2;

typename SizePolicy::RegSlotType R3;

typename SizePolicy::RegSlotType R4;

typename SizePolicy::RegSlotType R5;

typename SizePolicy::RegSlotType R6;

};

template <typename SizePolicy>

struct OpLayoutT\_AsmReg2IntConst1

{

typename SizePolicy::RegSlotType R0;

typename SizePolicy::RegSlotType R1;

typename int C2;

};

template <typename SizePolicy>

struct OpLayoutT\_Int1Double1

{

typename SizePolicy::RegSlotType I0;

typename SizePolicy::RegSlotType D1;

};

template <typename SizePolicy>

struct OpLayoutT\_Int1Float1

{

typename SizePolicy::RegSlotType I0;

typename SizePolicy::RegSlotType F1;

};

template <typename SizePolicy>

struct OpLayoutT\_Double1Int1

{

typename SizePolicy::RegSlotType D0;

typename SizePolicy::RegSlotType I1;

};

template <typename SizePolicy>

struct OpLayoutT\_Double1Float1

{

typename SizePolicy::RegSlotType D0;

typename SizePolicy::RegSlotType F1;

};

template <typename SizePolicy>

struct OpLayoutT\_Double1Reg1

{

typename SizePolicy::RegSlotType D0;

typename SizePolicy::RegSlotType R1;

};

template <typename SizePolicy>

struct OpLayoutT\_Float1Reg1

{

typename SizePolicy::RegSlotType F0;

typename SizePolicy::RegSlotType R1;

};

template <typename SizePolicy>

struct OpLayoutT\_Int1Reg1

{

typename SizePolicy::RegSlotType I0;

typename SizePolicy::RegSlotType R1;

};

template <typename SizePolicy>

struct OpLayoutT\_Reg1Double1

{

typename SizePolicy::RegSlotType R0;

typename SizePolicy::RegSlotType D1;

};

template <typename SizePolicy>

struct OpLayoutT\_Reg1Float1

{

typename SizePolicy::RegSlotType R0;

typename SizePolicy::RegSlotType F1;

};

template <typename SizePolicy>

struct OpLayoutT\_Reg1Int1

{

typename SizePolicy::RegSlotType R0;

typename SizePolicy::RegSlotType I1;

};

template <typename SizePolicy>

struct OpLayoutT\_Int1Double2

{

typename SizePolicy::RegSlotType I0;

typename SizePolicy::RegSlotType D1;

typename SizePolicy::RegSlotType D2;

};

template <typename SizePolicy>

struct OpLayoutT\_Int1Float2

{

typename SizePolicy::RegSlotType I0;

typename SizePolicy::RegSlotType F1;

typename SizePolicy::RegSlotType F2;

};

template <typename SizePolicy>

struct OpLayoutT\_Int2

{

typename SizePolicy::RegSlotType I0;

typename SizePolicy::RegSlotType I1;

};

template <typename SizePolicy>

struct OpLayoutT\_Int1Const1

{

typename SizePolicy::RegSlotType I0;

typename int C1;

};

template <typename SizePolicy>

struct OpLayoutT\_Int3

{

typename SizePolicy::RegSlotType I0;

typename SizePolicy::RegSlotType I1;

typename SizePolicy::RegSlotType I2;

};

template <typename SizePolicy>

struct OpLayoutT\_Double2

{

typename SizePolicy::RegSlotType D0;

typename SizePolicy::RegSlotType D1;

};

template <typename SizePolicy>

struct OpLayoutT\_Float2

{

typename SizePolicy::RegSlotType F0;

typename SizePolicy::RegSlotType F1;

};

template <typename SizePolicy>

struct OpLayoutT\_Float3

{

typename SizePolicy::RegSlotType F0;

typename SizePolicy::RegSlotType F1;

typename SizePolicy::RegSlotType F2;

};

template <typename SizePolicy>

struct OpLayoutT\_Float1Double1

{

typename SizePolicy::RegSlotType F0;

typename SizePolicy::RegSlotType D1;

};

template <typename SizePolicy>

struct OpLayoutT\_Float1Int1

{

typename SizePolicy::RegSlotType F0;

typename SizePolicy::RegSlotType I1;

};

template <typename SizePolicy>

struct OpLayoutT\_Double3

{

typename SizePolicy::RegSlotType D0;

typename SizePolicy::RegSlotType D1;

typename SizePolicy::RegSlotType D2;

};

template <typename SizePolicy>

struct OpLayoutT\_AsmUnsigned1

{

typename SizePolicy::UnsignedType C1;

};

struct OpLayoutAsmBr

{

int32 RelativeJumpOffset;

};

template <typename SizePolicy>

struct OpLayoutT\_BrInt1

{

int32 RelativeJumpOffset;

typename SizePolicy::RegSlotType I1;

};

template <typename SizePolicy>

struct OpLayoutT\_BrInt2

{

int32 RelativeJumpOffset;

typename SizePolicy::RegSlotType I1;

typename SizePolicy::RegSlotType I2;

};

/\* Float32x4 layouts \*/

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

template <typename SizePolicy>

struct OpLayoutT\_Float32x4\_2

{

typename SizePolicy::RegSlotType F4\_0;

typename SizePolicy::RegSlotType F4\_1;

};

template <typename SizePolicy>

struct OpLayoutT\_Float32x4\_3

{

typename SizePolicy::RegSlotType F4\_0;

typename SizePolicy::RegSlotType F4\_1;

typename SizePolicy::RegSlotType F4\_2;

};

template <typename SizePolicy>

struct OpLayoutT\_Float32x4\_4

{

typename SizePolicy::RegSlotType F4\_0;

typename SizePolicy::RegSlotType F4\_1;

typename SizePolicy::RegSlotType F4\_2;

typename SizePolicy::RegSlotType F4\_3;

};

// 4 floats -> float32x4.

template <typename SizePolicy>

struct OpLayoutT\_Float32x4\_1Float4

{

typename SizePolicy::RegSlotType F4\_0;

typename SizePolicy::RegSlotType F1;

typename SizePolicy::RegSlotType F2;

typename SizePolicy::RegSlotType F3;

typename SizePolicy::RegSlotType F4;

};

template <typename SizePolicy>

struct OpLayoutT\_Float32x4\_2Int4

{

typename SizePolicy::RegSlotType F4\_0;

typename SizePolicy::RegSlotType F4\_1;

typename SizePolicy::RegSlotType I2;

typename SizePolicy::RegSlotType I3;

typename SizePolicy::RegSlotType I4;

typename SizePolicy::RegSlotType I5;

};

template <typename SizePolicy>

struct OpLayoutT\_Float32x4\_3Int4

{

typename SizePolicy::RegSlotType F4\_0;

typename SizePolicy::RegSlotType F4\_1;

typename SizePolicy::RegSlotType F4\_2;

typename SizePolicy::RegSlotType I3;

typename SizePolicy::RegSlotType I4;

typename SizePolicy::RegSlotType I5;

typename SizePolicy::RegSlotType I6;

};

template <typename SizePolicy>

struct OpLayoutT\_Float32x4\_1Float1

{

typename SizePolicy::RegSlotType F4\_0;

typename SizePolicy::RegSlotType F1;

};

template <typename SizePolicy>

struct OpLayoutT\_Float32x4\_2Float1

{

typename SizePolicy::RegSlotType F4\_0;

typename SizePolicy::RegSlotType F4\_1;

typename SizePolicy::RegSlotType F2;

};

template <typename SizePolicy>

struct OpLayoutT\_Float32x4\_1Float64x2\_1

{

typename SizePolicy::RegSlotType F4\_0;

typename SizePolicy::RegSlotType D2\_1;

};

template <typename SizePolicy>

struct OpLayoutT\_Float32x4\_1Int32x4\_1

{

typename SizePolicy::RegSlotType F4\_0;

typename SizePolicy::RegSlotType I4\_1;

};

template <typename SizePolicy>

struct OpLayoutT\_Float32x4\_1Int32x4\_1Float32x4\_2

{

typename SizePolicy::RegSlotType F4\_0;

typename SizePolicy::RegSlotType I4\_1;

typename SizePolicy::RegSlotType F4\_2;

typename SizePolicy::RegSlotType F4\_3;

};

template <typename SizePolicy>

struct OpLayoutT\_Reg1Float32x4\_1

{

typename SizePolicy::RegSlotType R0;

typename SizePolicy::RegSlotType F4\_1;

};

template <typename SizePolicy>

struct OpLayoutT\_Int1Float32x4\_1

{

typename SizePolicy::RegSlotType I0;

typename SizePolicy::RegSlotType F4\_1;

};

/\* Int32x4 layouts \*/

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

template <typename SizePolicy>

struct OpLayoutT\_Int32x4\_2

{

typename SizePolicy::RegSlotType I4\_0;

typename SizePolicy::RegSlotType I4\_1;

};

template <typename SizePolicy>

struct OpLayoutT\_Int32x4\_3

{

typename SizePolicy::RegSlotType I4\_0;

typename SizePolicy::RegSlotType I4\_1;

typename SizePolicy::RegSlotType I4\_2;

};

template <typename SizePolicy>

struct OpLayoutT\_Int32x4\_4

{

typename SizePolicy::RegSlotType I4\_0;

typename SizePolicy::RegSlotType I4\_1;

typename SizePolicy::RegSlotType I4\_2;

typename SizePolicy::RegSlotType I4\_3;

};

template <typename SizePolicy>

struct OpLayoutT\_Int32x4\_1Int1

{

typename SizePolicy::RegSlotType I4\_0;

typename SizePolicy::RegSlotType I1;

};

template <typename SizePolicy>

struct OpLayoutT\_Int32x4\_1Int4

{

typename SizePolicy::RegSlotType I4\_0;

typename SizePolicy::RegSlotType I1;

typename SizePolicy::RegSlotType I2;

typename SizePolicy::RegSlotType I3;

typename SizePolicy::RegSlotType I4;

};

template <typename SizePolicy>

struct OpLayoutT\_Int32x4\_2Int4

{

typename SizePolicy::RegSlotType I4\_0;

typename SizePolicy::RegSlotType I4\_1;

typename SizePolicy::RegSlotType I2;

typename SizePolicy::RegSlotType I3;

typename SizePolicy::RegSlotType I4;

typename SizePolicy::RegSlotType I5;

};

template <typename SizePolicy>

struct OpLayoutT\_Int32x4\_3Int4

{

typename SizePolicy::RegSlotType I4\_0;

typename SizePolicy::RegSlotType I4\_1;

typename SizePolicy::RegSlotType I4\_2;

typename SizePolicy::RegSlotType I3;

typename SizePolicy::RegSlotType I4;

typename SizePolicy::RegSlotType I5;

typename SizePolicy::RegSlotType I6;

};

template <typename SizePolicy>

struct OpLayoutT\_Int32x4\_2Int1

{

typename SizePolicy::RegSlotType I4\_0;

typename SizePolicy::RegSlotType I4\_1;

typename SizePolicy::RegSlotType I2;

};

template <typename SizePolicy>

struct OpLayoutT\_Int32x4\_2Int2

{

typename SizePolicy::RegSlotType I4\_0;

typename SizePolicy::RegSlotType I4\_1;

typename SizePolicy::RegSlotType I2;

typename SizePolicy::RegSlotType I3;

};

template <typename SizePolicy>

struct OpLayoutT\_Int1Int32x4\_1Int1

{

typename SizePolicy::RegSlotType I0;

typename SizePolicy::RegSlotType I4\_1;

typename SizePolicy::RegSlotType I2;

};

template <typename SizePolicy>

struct OpLayoutT\_Float32x4\_2Int1Float1

{

typename SizePolicy::RegSlotType F4\_0;

typename SizePolicy::RegSlotType F4\_1;

typename SizePolicy::RegSlotType I2;

typename SizePolicy::RegSlotType F3;

};

template <typename SizePolicy>

struct OpLayoutT\_Float1Float32x4\_1Int1

{

typename SizePolicy::RegSlotType F0;

typename SizePolicy::RegSlotType F4\_1;

typename SizePolicy::RegSlotType I2;

};

template <typename SizePolicy>

struct OpLayoutT\_Reg1Int32x4\_1

{

typename SizePolicy::RegSlotType R0;

typename SizePolicy::RegSlotType I4\_1;

};

template <typename SizePolicy>

struct OpLayoutT\_Int32x4\_1Float64x2\_1

{

typename SizePolicy::RegSlotType I4\_0;

typename SizePolicy::RegSlotType D2\_1;

};

template <typename SizePolicy>

struct OpLayoutT\_Int32x4\_1Float32x4\_1

{

typename SizePolicy::RegSlotType I4\_0;

typename SizePolicy::RegSlotType F4\_1;

};

template <typename SizePolicy>

struct OpLayoutT\_Int1Int32x4\_1

{

typename SizePolicy::RegSlotType I0;

typename SizePolicy::RegSlotType I4\_1;

};

/\* Float64x2 layouts \*/

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

template <typename SizePolicy>

struct OpLayoutT\_Float64x2\_2

{

typename SizePolicy::RegSlotType D2\_0;

typename SizePolicy::RegSlotType D2\_1;

};

template <typename SizePolicy>

struct OpLayoutT\_Float64x2\_3

{

typename SizePolicy::RegSlotType D2\_0;

typename SizePolicy::RegSlotType D2\_1;

typename SizePolicy::RegSlotType D2\_2;

};

template <typename SizePolicy>

struct OpLayoutT\_Float64x2\_4

{

typename SizePolicy::RegSlotType D2\_0;

typename SizePolicy::RegSlotType D2\_1;

typename SizePolicy::RegSlotType D2\_2;

typename SizePolicy::RegSlotType D2\_3;

};

template <typename SizePolicy>

struct OpLayoutT\_Float64x2\_1Double2

{

typename SizePolicy::RegSlotType D2\_0;

typename SizePolicy::RegSlotType D1;

typename SizePolicy::RegSlotType D2;

};

template <typename SizePolicy>

struct OpLayoutT\_Float64x2\_1Double1

{

typename SizePolicy::RegSlotType D2\_0;

typename SizePolicy::RegSlotType D1;

};

template <typename SizePolicy>

struct OpLayoutT\_Float64x2\_2Double1

{

typename SizePolicy::RegSlotType D2\_0;

typename SizePolicy::RegSlotType D2\_1;

typename SizePolicy::RegSlotType D2;

};

template <typename SizePolicy>

struct OpLayoutT\_Float64x2\_2Int2

{

typename SizePolicy::RegSlotType D2\_0;

typename SizePolicy::RegSlotType D2\_1;

typename SizePolicy::RegSlotType I2;

typename SizePolicy::RegSlotType I3;

};

template <typename SizePolicy>

struct OpLayoutT\_Float64x2\_3Int2

{

typename SizePolicy::RegSlotType D2\_0;

typename SizePolicy::RegSlotType D2\_1;

typename SizePolicy::RegSlotType D2\_2;

typename SizePolicy::RegSlotType I3;

typename SizePolicy::RegSlotType I4;

};

template <typename SizePolicy>

struct OpLayoutT\_Float64x2\_1Float32x4\_1

{

typename SizePolicy::RegSlotType D2\_0;

typename SizePolicy::RegSlotType F4\_1;

};

template <typename SizePolicy>

struct OpLayoutT\_Float64x2\_1Int32x4\_1

{

typename SizePolicy::RegSlotType D2\_0;

typename SizePolicy::RegSlotType I4\_1;

};

template <typename SizePolicy>

struct OpLayoutT\_Float64x2\_1Int32x4\_1Float64x2\_2

{

typename SizePolicy::RegSlotType D2\_0;

typename SizePolicy::RegSlotType I4\_1;

typename SizePolicy::RegSlotType D2\_2;

typename SizePolicy::RegSlotType D2\_3;

};

template <typename SizePolicy>

struct OpLayoutT\_Reg1Float64x2\_1

{

typename SizePolicy::RegSlotType R0;

typename SizePolicy::RegSlotType D2\_1;

};

template <typename SizePolicy>

struct OpLayoutT\_Int1Float64x2\_1

{

typename SizePolicy::RegSlotType I0;

typename SizePolicy::RegSlotType D2\_1;

};

template <typename SizePolicy>

struct OpLayoutT\_AsmSimdTypedArr

{

// force encode 4 bytes because it can be a value

uint32 SlotIndex;

typename SizePolicy::RegSlotType Value;

int8 ViewType;

int8 DataWidth; // # of bytes to load/store

};

// Generate the multi size layout type defs

#define LAYOUT\_TYPE\_WMS(layout) \

typedef OpLayoutT\_##layout<LargeLayoutSizePolicy> OpLayout##layout##\_Large; \

typedef OpLayoutT\_##layout<MediumLayoutSizePolicy> OpLayout##layout##\_Medium; \

typedef OpLayoutT\_##layout<SmallLayoutSizePolicy> OpLayout##layout##\_Small;

#include "LayoutTypesAsmJs.h"

#pragma pack(pop)

// Generate structure to automatically map layout to its info

template <OpLayoutTypeAsmJs::\_E layout> struct OpLayoutInfoAsmJs;

#define LAYOUT\_TYPE(layout) \

CompileAssert(sizeof(OpLayout##layout) <= MaxLayoutSize); \

template <> struct OpLayoutInfoAsmJs<OpLayoutTypeAsmJs::layout> \

{ \

static const bool HasMultiSizeLayout = false; \

};

#define LAYOUT\_TYPE\_WMS(layout) \

CompileAssert(sizeof(OpLayout##layout##\_Large) <= MaxLayoutSize); \

template <> struct OpLayoutInfoAsmJs<OpLayoutTypeAsmJs::layout> \

{ \

static const bool HasMultiSizeLayout = true; \

};

#include "LayoutTypesAsmJs.h"

// Generate structure to automatically map opcode to its info

// Also generate assert to make sure the layout and opcode use the same macro with and without multiple size layout

template <OpCodeAsmJs opcode> struct OpCodeInfoAsmJs;

#define DEFINE\_OPCODEINFO(op, layout, extended) \

CompileAssert(!OpLayoutInfoAsmJs<OpLayoutTypeAsmJs::layout>::HasMultiSizeLayout); \

template <> struct OpCodeInfoAsmJs<OpCodeAsmJs::op> \

{ \

static const OpLayoutTypeAsmJs::\_E Layout = OpLayoutTypeAsmJs::layout; \

static const bool HasMultiSizeLayout = false; \

static const bool IsExtendedOpcode = extended; \

typedef OpLayout##layout LayoutType; \

};

#define DEFINE\_OPCODEINFO\_WMS(op, layout, extended) \

CompileAssert(OpLayoutInfoAsmJs<OpLayoutTypeAsmJs::layout>::HasMultiSizeLayout); \

template <> struct OpCodeInfoAsmJs<OpCodeAsmJs::op> \

{ \

static const OpLayoutTypeAsmJs::\_E Layout = OpLayoutTypeAsmJs::layout; \

static const bool HasMultiSizeLayout = true; \

static const bool IsExtendedOpcode = extended; \

typedef OpLayout##layout##\_Large LayoutType\_Large; \

typedef OpLayout##layout##\_Medium LayoutType\_Medium; \

typedef OpLayout##layout##\_Small LayoutType\_Small; \

};

#define MACRO(op, layout, ...) DEFINE\_OPCODEINFO(op, layout, false)

#define MACRO\_WMS(op, layout, ...) DEFINE\_OPCODEINFO\_WMS(op, layout, false)

#define MACRO\_EXTEND(op, layout, ...) DEFINE\_OPCODEINFO(op, layout, true)

#define MACRO\_EXTEND\_WMS(op, layout, ...) DEFINE\_OPCODEINFO\_WMS(op, layout, true)

#include "OpCodesAsmJs.h"

#undef DEFINE\_OPCODEINFO

#undef DEFINE\_OPCODEINFO\_WMS

}

#endif

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

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

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

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

#pragma once

namespace Js {

typedef uint8 ArgSlot\_OneByte;

typedef uint32 CacheId;

typedef uint8 CacheId\_OneByte;

typedef uint16 CacheId\_TwoByte;

typedef uint32 RootCacheId;

typedef uint16 PropertyIdIndexType\_TwoByte;

typedef uint32 PropertyIdIndexType;

#ifdef BYTECODE\_BRANCH\_ISLAND

typedef int16 JumpOffset;

typedef int32 LongJumpOffset;

#else

typedef int32 JumpOffset;

#endif

// This is used to estimate when we need to emit long branches

uint const MaxLayoutSize = 28; // Increase this when we see larger layout

uint const MaxOpCodeSize = 2;

enum LayoutSize

{

SmallLayout,

MediumLayout,

LargeLayout,

};

template <LayoutSize layoutSize>

struct LayoutSizePolicy;

template <>

struct LayoutSizePolicy<LargeLayout>

{

typedef RegSlot RegSlotType;

typedef RegSlot RegSlotSType;

typedef ArgSlot ArgSlotType;

typedef CacheId CacheIdType;

typedef PropertyIdIndexType PropertyIdIndexType;

typedef uint32 UnsignedType;

static const LayoutSize LayoutEnum = LargeLayout;

template <typename T>

static bool Assign(T& dst, T src) { dst = src; return true; }

};

typedef LayoutSizePolicy<LargeLayout> LargeLayoutSizePolicy;

template <>

struct LayoutSizePolicy<SmallLayout>

{

typedef RegSlot\_OneByte RegSlotType;

typedef RegSlot\_OneSByte RegSlotSType;

typedef ArgSlot\_OneByte ArgSlotType;

typedef CacheId\_OneByte CacheIdType;

typedef PropertyIdIndexType\_TwoByte PropertyIdIndexType;

typedef byte UnsignedType;

static const LayoutSize LayoutEnum = SmallLayout;

template <typename T1, typename T2>

static bool Assign(T1& dst, T2 src)

{

#ifdef BYTECODE\_TESTING

if (Configuration::Global.flags.LargeByteCodeLayout

|| Configuration::Global.flags.MediumByteCodeLayout)

{

return false;

}

#endif

dst = (T1)src;

return ((T2)dst == src);

}

};

typedef LayoutSizePolicy<SmallLayout> SmallLayoutSizePolicy;

template <>

struct LayoutSizePolicy<MediumLayout>

{

typedef RegSlot\_TwoByte RegSlotType;

typedef RegSlot\_TwoSByte RegSlotSType;

typedef ArgSlot\_OneByte ArgSlotType;

typedef CacheId\_TwoByte CacheIdType;

typedef PropertyIdIndexType\_TwoByte PropertyIdIndexType;

typedef uint16 UnsignedType;

static const LayoutSize LayoutEnum = MediumLayout;

template <typename T1, typename T2>

static bool Assign(T1& dst, T2 src)

{

#ifdef BYTECODE\_TESTING

if (Configuration::Global.flags.LargeByteCodeLayout)

{

return false;

}

#endif

dst = (T1)src;

return ((T2)dst == src);

}

};

typedef LayoutSizePolicy<MediumLayout> MediumLayoutSizePolicy;

struct OpLayoutEmpty

{

// Although empty structs are one byte, the Empty layout are not written out.

};

} // 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.

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

#pragma once

namespace Js

{

struct PropertyIdArray

{

uint32 count;

bool hadDuplicates;

bool has\_\_proto\_\_; // Only used for object literal

bool hasNonSimpleParams;

PropertyId elements[];

PropertyIdArray(uint32 count, bool hadDuplicates = false, bool has\_\_proto\_\_ = false, bool hasNonSimpleParams = false) : count(count), hadDuplicates(hadDuplicates), has\_\_proto\_\_(has\_\_proto\_\_), hasNonSimpleParams(hasNonSimpleParams)

{

}

size\_t GetDataSize(uint32 extraSlots) const { return sizeof(PropertyIdArray) + sizeof(PropertyId) \* (count + extraSlots); }

};

};

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

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

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

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

#include "RuntimeByteCodePch.h"

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

// 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

#include "Parser.h"

#include "Runtime.h"

#include "ByteCode\Symbol.h"

#include "ByteCode\Scope.h"

#include "ByteCode\FuncInfo.h"

#include "ByteCode\ScopeInfo.h"

#include "ByteCode\StatementReader.h"

#include "ByteCode\ByteCodeDumper.h"

#include "ByteCode\ByteCodeWriter.h"

#include "ByteCode\ByteCodeGenerator.h"

#include "ByteCode\OpCodeUtilAsmJs.h"

#include "Language\AsmJsTypes.h"

#include "ByteCode\ByteCodeAPI.h"

#include "ByteCode\BackEndOpcodeAttr.h"

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

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

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

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

#include "RuntimeByteCodePch.h"

bool Scope::IsGlobalEvalBlockScope() const

{

return this->scopeType == ScopeType\_GlobalEvalBlock;

}

bool Scope::IsBlockScope(FuncInfo \*funcInfo)

{

return this != funcInfo->GetBodyScope() && this != funcInfo->GetParamScope();

}

void Scope::SetHasLocalInClosure(bool has)

{

// (Note: if any catch var is closure-captured, we won't merge the catch scope with the function scope.

// So don't mark the function scope "has local in closure".)

if (has && (this == func->GetBodyScope() || this == func->GetParamScope()) || (GetCanMerge() && (this->scopeType != ScopeType\_Catch && this->scopeType != ScopeType\_CatchParamPattern)))

{

func->SetHasLocalInClosure(true);

}

else

{

if (hasCrossScopeFuncAssignment)

{

func->SetHasMaybeEscapedNestedFunc(DebugOnly(L"InstantiateScopeWithCrossScopeAssignment"));

}

SetMustInstantiate(true);

}

}

int Scope::AddScopeSlot()

{

int slot = scopeSlotCount++;

if (scopeSlotCount == Js::ScopeSlots::MaxEncodedSlotCount)

{

this->GetEnclosingFunc()->SetHasMaybeEscapedNestedFunc(DebugOnly(L"TooManySlots"));

}

return slot;

}

void Scope::ForceAllSymbolNonLocalReference(ByteCodeGenerator \*byteCodeGenerator)

{

this->ForEachSymbol([this, byteCodeGenerator](Symbol \*const sym)

{

if (!sym->GetIsArguments())

{

sym->SetHasNonLocalReference(true, byteCodeGenerator);

this->GetFunc()->SetHasLocalInClosure(true);

}

});

}

bool Scope::IsEmpty() const

{

if (GetFunc()->bodyScope == this || (GetFunc()->IsGlobalFunction() && this->IsGlobalEvalBlockScope()))

{

return Count() == 0 && !GetFunc()->isThisLexicallyCaptured;

}

else

{

return Count() == 0;

}

}

void Scope::SetIsObject()

{

if (this->isObject)

{

return;

}

this->isObject = true;

// We might set the scope to be object after we have process the symbol

// (e.g. "With" scope referencing a symbol in an outer scope).

// If we have func assignment, we need to mark the function to not do stack nested function

// as these are now assigned to a scope object.

FuncInfo \* funcInfo = this->GetFunc();

if (funcInfo && !funcInfo->HasMaybeEscapedNestedFunc())

{

this->ForEachSymbolUntil([funcInfo](Symbol \* const sym)

{

if (sym->GetHasFuncAssignment())

{

funcInfo->SetHasMaybeEscapedNestedFunc(DebugOnly(L"DelayedObjectScopeAssignment"));

return true;

}

return false;

});

}

}

void Scope::MergeParamAndBodyScopes(ParseNode \*pnodeScope, ByteCodeGenerator \*byteCodeGenerator)

{

Assert(pnodeScope->sxFnc.funcInfo);

Scope \*paramScope = pnodeScope->sxFnc.pnodeScopes->sxBlock.scope;

Scope \*bodyScope = pnodeScope->sxFnc.pnodeBodyScope->sxBlock.scope;

Assert(paramScope->m\_symList == nullptr || paramScope->symbolTable == nullptr);

Assert(bodyScope->m\_symList == nullptr || bodyScope->symbolTable == nullptr);

if (paramScope->Count() == 0)

{

// Once the scopes are merged, there's no reason to instantiate the param scope.

paramScope->SetMustInstantiate(false);

// Scopes are already merged or we don't have an arguments object. Go ahead and

// remove the param scope from the scope chain.

bodyScope->SetEnclosingScope(paramScope->GetEnclosingScope());

return;

}

bodyScope->ForEachSymbol([&](Symbol \* sym)

{

// Duplicate 'arguments' - param scope arguments wins.

if (byteCodeGenerator->UseParserBindings()

&& sym->GetDecl()->sxVar.pid == byteCodeGenerator->GetParser()->names()->arguments)

{

return;

}

Assert(paramScope->m\_symList == nullptr || paramScope->FindLocalSymbol(sym->GetName()) == nullptr);

paramScope->AddNewSymbol(sym);

});

// Reassign non-formal slot positions. Formals need to keep their slot positions to ensure

// the argument object works properly. Other symbols need to be reassigned slot positions.

paramScope->ForEachSymbol([&](Symbol \* sym)

{

if (sym->GetSymbolType() != STFormal && sym->GetScopeSlot() != Js::Constants::NoProperty)

{

sym->SetScopeSlot(Js::Constants::NoProperty);

sym->EnsureScopeSlot(pnodeScope->sxFnc.funcInfo);

}

sym->SetScope(bodyScope);

});

bodyScope->m\_count = paramScope->m\_count;

bodyScope->m\_symList = paramScope->m\_symList;

bodyScope->scopeSlotCount = paramScope->scopeSlotCount;

if (bodyScope->symbolTable != nullptr)

{

Adelete(byteCodeGenerator->GetAllocator(), bodyScope->symbolTable);

bodyScope->symbolTable = nullptr;

}

bodyScope->symbolTable = paramScope->symbolTable;

if (paramScope->GetIsObject())

{

bodyScope->SetIsObject();

}

if (paramScope->GetMustInstantiate())

{

bodyScope->SetMustInstantiate(true);

}

// Once the scopes are merged, there's no reason to instantiate the param scope.

paramScope->SetMustInstantiate(false);

paramScope->m\_count = 0;

paramScope->scopeSlotCount = 0;

paramScope->m\_symList = nullptr;

paramScope->symbolTable = nullptr;

// Remove the parameter scope from the scope chain.

bodyScope->SetEnclosingScope(paramScope->GetEnclosingScope());

}

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

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

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

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

enum ScopeType

{

ScopeType\_Unknown,

ScopeType\_Global,

ScopeType\_GlobalEvalBlock,

ScopeType\_FunctionBody,

ScopeType\_FuncExpr,

ScopeType\_Block,

ScopeType\_Catch,

ScopeType\_CatchParamPattern,

ScopeType\_With,

ScopeType\_Parameter

};

class Scope

{

private:

Scope \*enclosingScope;

Js::RegSlot location;

FuncInfo \*func;

SymbolTable \*symbolTable;

Symbol \*m\_symList;

int m\_count;

ArenaAllocator \*alloc;

uint scopeSlotCount; // count of slots in the local scope

uint innerScopeIndex;

ScopeType const scopeType;

BYTE isDynamic : 1;

BYTE isObject : 1;

BYTE canMerge : 1;

BYTE capturesAll : 1;

BYTE mustInstantiate : 1;

BYTE hasCrossScopeFuncAssignment : 1;

public:

#if DBG

BYTE isRestored : 1;

#endif

Scope(ArenaAllocator \*alloc, ScopeType scopeType, bool useSymbolTable = false, int capacity = 0) :

alloc(alloc),

func(nullptr),

enclosingScope(nullptr),

isDynamic(false),

isObject(false),

canMerge(true),

capturesAll(false),

mustInstantiate(false),

hasCrossScopeFuncAssignment(false),

location(Js::Constants::NoRegister),

symbolTable(nullptr),

m\_symList(nullptr),

m\_count(0),

scopeSlotCount(0),

innerScopeIndex((uint)-1),

scopeType(scopeType)

#if DBG

, isRestored(false)

#endif

{

if (useSymbolTable)

{

symbolTable = Anew(alloc, SymbolTable, alloc, capacity);

}

}

~Scope()

{

if (symbolTable)

{

Adelete(alloc, symbolTable);

symbolTable = nullptr;

}

}

Symbol \*FindLocalSymbol(SymbolName const& key)

{

Symbol \*sym = nullptr;

if (symbolTable)

{

return symbolTable->Lookup(key);

}

for (sym = m\_symList; sym; sym = sym->GetNext())

{

if (sym->GetName() == key)

{

break;

}

}

return sym;

}

template<class Fn>

void ForEachSymbol(Fn fn)

{

if (symbolTable)

{

symbolTable->Map(fn);

}

else

{

for (Symbol \*sym = m\_symList; sym;)

{

Symbol \*next = sym->GetNext();

fn(sym);

sym = next;

}

}

}

template<class Fn>

void ForEachSymbolUntil(Fn fn)

{

if (symbolTable)

{

symbolTable->MapUntil(fn);

}

else

{

for (Symbol \*sym = m\_symList; sym;)

{

Symbol \*next = sym->GetNext();

if (fn(sym))

{

return;

}

sym = next;

}

}

}

// For JScript, this should not return NULL because

// there is always an enclosing global scope.

Symbol \*FindSymbol(SymbolName const& name, SymbolType symbolType, bool fCreate = true)

{

Symbol \*sym = FindLocalSymbol(name);

if (sym == nullptr)

{

if (enclosingScope != nullptr)

{

sym = enclosingScope->FindSymbol(name, symbolType);

}

else if (fCreate)

{

sym = Anew(alloc, Symbol, name, nullptr, symbolType);

AddNewSymbol(sym);

}

}

return sym;

}

void AddSymbol(Symbol \*sym)

{

if (enclosingScope == nullptr)

{

sym->SetIsGlobal(true);

}

sym->SetScope(this);

if (symbolTable)

{

symbolTable->AddNew(sym);

}

else

{

for (Symbol \*symInList = m\_symList; symInList; symInList = symInList->GetNext())

{

if (symInList->GetName() == sym->GetName())

{

return;

}

}

sym->SetNext(m\_symList);

m\_symList = sym;

m\_count++;

}

}

void AddNewSymbol(Symbol \*sym)

{

if (scopeType == ScopeType\_Global)

{

sym->SetIsGlobal(true);

}

sym->SetScope(this);

if (symbolTable)

{

symbolTable->Add(sym);

}

else

{

sym->SetNext(m\_symList);

m\_symList = sym;

m\_count++;

}

}

bool HasStaticPathToAncestor(Scope const \* target) const

{

return target == this || (!isDynamic && enclosingScope != nullptr && enclosingScope->HasStaticPathToAncestor(target));

}

void SetEnclosingScope(Scope \*enclosingScope)

{

// Check for scope cycles

Assert(enclosingScope != this);

Assert(enclosingScope == nullptr || this != enclosingScope->GetEnclosingScope());

this->enclosingScope = enclosingScope;

}

Scope \*GetEnclosingScope() const

{

return enclosingScope;

}

ScopeType GetScopeType() const

{

return this->scopeType;

}

bool IsInnerScope() const

{

return scopeType == ScopeType\_Block || scopeType == ScopeType\_Catch || scopeType == ScopeType\_CatchParamPattern || scopeType == ScopeType\_GlobalEvalBlock;

}

int Count() const

{

if (symbolTable)

{

return symbolTable->Count();

}

return m\_count;

}

void SetFunc(FuncInfo \*func)

{

this->func = func;

}

FuncInfo \*GetFunc() const

{

return func;

}

FuncInfo \*GetEnclosingFunc()

{

Scope \*scope = this;

while (scope && scope->func == nullptr)

{

scope = scope->GetEnclosingScope();

}

AnalysisAssert(scope);

return scope->func;

}

void SetLocation(Js::RegSlot loc) { location = loc; }

Js::RegSlot GetLocation() const { return location; }

void SetIsDynamic(bool is) { isDynamic = is; }

bool GetIsDynamic() const { return isDynamic; }

bool IsEmpty() const;

bool IsBlockScope(FuncInfo \*funcInfo);

void SetIsObject();

bool GetIsObject() const { return isObject; }

void SetCapturesAll(bool does) { capturesAll = does; }

bool GetCapturesAll() const { return capturesAll; }

void SetMustInstantiate(bool must) { mustInstantiate = must; }

bool GetMustInstantiate() const { return mustInstantiate; }

void SetCanMerge(bool can) { canMerge = can; }

bool GetCanMerge() const { return canMerge && !mustInstantiate && !isObject; }

void SetScopeSlotCount(uint i) { scopeSlotCount = i; }

uint GetScopeSlotCount() const { return scopeSlotCount; }

void SetHasLocalInClosure(bool has);

bool HasInnerScopeIndex() const { return innerScopeIndex != (uint)-1; }

uint GetInnerScopeIndex() const { return innerScopeIndex; }

void SetInnerScopeIndex(uint index) { Assert(innerScopeIndex == (uint)-1 || innerScopeIndex == index); innerScopeIndex = index; }

int AddScopeSlot();

void SetHasCrossScopeFuncAssignment() { hasCrossScopeFuncAssignment = true; }

void ForceAllSymbolNonLocalReference(ByteCodeGenerator \*byteCodeGenerator);

bool IsGlobalEvalBlockScope() const;

static void MergeParamAndBodyScopes(ParseNode \*pnodeScope, ByteCodeGenerator \* byteCodeGenerator);

};

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

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

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

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

#include "RuntimeByteCodePch.h"

namespace Js

{

//

// Persist one symbol info into ScopeInfo.

//

void ScopeInfo::SaveSymbolInfo(Symbol\* sym, MapSymbolData\* mapSymbolData)

{

// We don't need to create slot for or save "arguments"

if (!sym->GetIsArguments()

// Function expression may not have nonLocalReference, exclude them.

&& (!sym->GetFuncExpr() || sym->GetHasNonLocalReference()))

{

// Any symbol may have non-local ref from deferred child. Allocate slot for it.

Assert(sym->GetHasNonLocalReference());

Js::PropertyId scopeSlot = sym->EnsureScopeSlot(mapSymbolData->func);

Js::PropertyId propertyId = sym->EnsurePosition(mapSymbolData->func);

this->SetSymbolId(scopeSlot, propertyId);

this->SetSymbolType(scopeSlot, sym->GetSymbolType());

this->SetHasFuncAssignment(scopeSlot, sym->GetHasFuncAssignment());

this->SetIsBlockVariable(scopeSlot, sym->GetIsBlockVar());

}

TRACE\_BYTECODE(L"%12s %d\n", sym->GetName().GetBuffer(), sym->GetScopeSlot());

}

//

// Create scope info for a deferred child to refer to its parent ParseableFunctionInfo.

//

ScopeInfo\* ScopeInfo::FromParent(FunctionBody\* parent)

{

return RecyclerNew(parent->GetScriptContext()->GetRecycler(), // Alloc with ParseableFunctionInfo

ScopeInfo, parent, 0);

}

inline void AddSlotCount(int& count, int addCount)

{

if (addCount != 0 && Int32Math::Add(count, addCount, &count))

{

::Math::DefaultOverflowPolicy();

}

}

//

// Create scope info for an outer scope.

//

ScopeInfo\* ScopeInfo::FromScope(ByteCodeGenerator\* byteCodeGenerator, FunctionBody\* parent, Scope\* scope, ScriptContext \*scriptContext)

{

int count = scope->Count();

// Add same name args place holder slot counts

AddSlotCount(count, scope->GetFunc()->sameNameArgsPlaceHolderSlotCount);

AddSlotCount(count, scope->GetFunc()->thisScopeSlot != Js::Constants::NoRegister ? 1 : 0);

AddSlotCount(count, scope->GetFunc()->newTargetScopeSlot != Js::Constants::NoRegister ? 1 : 0);

ScopeInfo\* scopeInfo = RecyclerNewPlusZ(scriptContext->GetRecycler(),

count \* sizeof(SymbolInfo),

ScopeInfo, parent, count);

scopeInfo->isDynamic = scope->GetIsDynamic();

scopeInfo->isObject = scope->GetIsObject();

scopeInfo->mustInstantiate = scope->GetMustInstantiate();

scopeInfo->isCached = (scope->GetFunc()->GetBodyScope() == scope) && scope->GetFunc()->GetHasCachedScope();

scopeInfo->isGlobalEval = scope->GetScopeType() == ScopeType\_GlobalEvalBlock;

TRACE\_BYTECODE(L"\nSave ScopeInfo: %s parent: %s #symbols: %d %s\n",

scope->GetFunc()->name, parent->GetDisplayName(), count, scopeInfo->isObject ? L"isObject" : L"");

MapSymbolData mapSymbolData = { byteCodeGenerator, scope->GetFunc() };

scope->ForEachSymbol([&mapSymbolData, scopeInfo, scope](Symbol \* sym)

{

Assert(scope == sym->GetScope());

scopeInfo->SaveSymbolInfo(sym, &mapSymbolData);

});

return scopeInfo;

}

//

// Clone a ScopeInfo object

//

ScopeInfo \*ScopeInfo::CloneFor(ParseableFunctionInfo \*body)

{

auto count = this->symbolCount;

auto symbolsSize = count \* sizeof(SymbolInfo);

auto scopeInfo = RecyclerNewPlusZ(parent->GetScriptContext()->GetRecycler(), symbolsSize,

ScopeInfo, parent, count);

scopeInfo->isDynamic = this->isDynamic;

scopeInfo->isObject = this->isObject;

scopeInfo->mustInstantiate = this->mustInstantiate;

scopeInfo->isCached = this->isCached;

scopeInfo->isGlobalEval = this->isGlobalEval;

if (funcExprScopeInfo)

{

scopeInfo->funcExprScopeInfo = funcExprScopeInfo->CloneFor(body);

}

if (paramScopeInfo)

{

scopeInfo->paramScopeInfo = paramScopeInfo->CloneFor(body);

}

memcpy\_s(scopeInfo->symbols, symbolsSize, this->symbols, symbolsSize);

return scopeInfo;

}

//

// Ensure the pids referenced by this scope are tracked.

//

void ScopeInfo::EnsurePidTracking(ScriptContext\* scriptContext)

{

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

{

auto propertyName = scriptContext->GetPropertyName(symbols[i].propertyId);

scriptContext->TrackPid(propertyName);

}

if (funcExprScopeInfo)

{

funcExprScopeInfo->EnsurePidTracking(scriptContext);

}

if (paramScopeInfo)

{

paramScopeInfo->EnsurePidTracking(scriptContext);

}

}

//

// Save needed scope info for a deferred child func. The scope info is empty and only links to parent.

//

void ScopeInfo::SaveParentScopeInfo(FuncInfo\* parentFunc, FuncInfo\* func)

{

Assert(func->IsDeferred());

// Parent must be parsed

FunctionBody\* parent = parentFunc->byteCodeFunction->GetFunctionBody();

ParseableFunctionInfo\* funcBody = func->byteCodeFunction;

TRACE\_BYTECODE(L"\nSave ScopeInfo: %s parent: %s\n\n",

funcBody->GetDisplayName(), parent->GetDisplayName());

funcBody->SetScopeInfo(FromParent(parent));

}

//

// Save scope info for an outer func which has deferred child.

//

void ScopeInfo::SaveScopeInfo(ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* parentFunc, FuncInfo\* func)

{

ParseableFunctionInfo\* funcBody = func->byteCodeFunction;

Assert((!func->IsGlobalFunction() || byteCodeGenerator->GetFlags() & fscrEvalCode) &&

(func->HasDeferredChild() || (funcBody->IsReparsed())));

// If we are reparsing a deferred function, we already have correct "parent" info in

// funcBody->scopeInfo. parentFunc is the knopProg shell and should not be used in this

// case. We should use existing parent if available.

FunctionBody \* parent = funcBody->GetScopeInfo() ?

funcBody->GetScopeInfo()->GetParent() :

parentFunc ? parentFunc->byteCodeFunction->GetFunctionBody() : nullptr;

ScopeInfo\* funcExprScopeInfo = nullptr;

Scope\* funcExprScope = func->GetFuncExprScope();

if (funcExprScope && funcExprScope->GetMustInstantiate())

{

funcExprScopeInfo = FromScope(byteCodeGenerator, parent, funcExprScope, funcBody->GetScriptContext());

}

Scope\* bodyScope = func->IsGlobalFunction() ? func->GetGlobalEvalBlockScope() : func->GetBodyScope();

ScopeInfo\* paramScopeInfo = nullptr;

Scope\* paramScope = func->GetParamScope();

if (paramScope && bodyScope->GetMustInstantiate())

{

paramScopeInfo = FromScope(byteCodeGenerator, parent, paramScope, funcBody->GetScriptContext());

}

ScopeInfo\* scopeInfo = FromScope(byteCodeGenerator, parent, bodyScope, funcBody->GetScriptContext());

scopeInfo->SetFuncExprScopeInfo(funcExprScopeInfo);

scopeInfo->SetParamScopeInfo(paramScopeInfo);

funcBody->SetScopeInfo(scopeInfo);

}

void ScopeInfo::SaveScopeInfoForDeferParse(ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* parentFunc, FuncInfo\* funcInfo)

{

// TODO: Not technically necessary, as we always do scope look up on eval if it is

// not found in the scope chain, and block scopes are always objects in eval.

// But if we save the global eval block scope for deferred child so that we can look up

// let/const in that scope with slot index instead of doing a scope lookup.

// We will have to implement encoding block scope info to enable, which will also

// enable defer parsing function that are in block scopes.

Assert(byteCodeGenerator->GetCurrentScope() == funcInfo->GetBodyScope());

if (funcInfo->IsDeferred())

{

// Don't need to remember the parent function if we have a global function

if (!parentFunc->IsGlobalFunction() ||

((byteCodeGenerator->GetFlags() & fscrEvalCode) && parentFunc->HasDeferredChild()))

{

// TODO: currently we only support defer nested function that is in function scope (no block scope, no with scope, etc.)

#if DBG

if (funcInfo->GetFuncExprScope() && funcInfo->GetFuncExprScope()->GetIsObject())

{

Assert(byteCodeGenerator->GetCurrentScope()->GetEnclosingScope() == funcInfo->GetFuncExprScope() &&

byteCodeGenerator->GetCurrentScope()->GetEnclosingScope()->GetEnclosingScope() ==

(parentFunc->IsGlobalFunction() ? parentFunc->GetGlobalEvalBlockScope() : parentFunc->GetBodyScope()));

}

else

{

Assert(byteCodeGenerator->GetCurrentScope()->GetEnclosingScope() ==

(parentFunc->IsGlobalFunction() ? parentFunc->GetGlobalEvalBlockScope() : parentFunc->GetBodyScope()));

}

#endif

Js::ScopeInfo::SaveParentScopeInfo(parentFunc, funcInfo);

}

}

else if (funcInfo->HasDeferredChild() ||

(!funcInfo->IsGlobalFunction() &&

funcInfo->byteCodeFunction &&

funcInfo->byteCodeFunction->IsReparsed() &&

funcInfo->byteCodeFunction->GetFunctionBody()->HasAllNonLocalReferenced()))

{

// When we reparse due to attach, we would need to capture all of them, since they were captured before going to debug mode.

Js::ScopeInfo::SaveScopeInfo(byteCodeGenerator, parentFunc, funcInfo);

}

}

//

// Load persisted scope info.

//

void ScopeInfo::GetScopeInfo(Parser \*parser, ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* funcInfo, Scope\* scope)

{

ScriptContext\* scriptContext;

ArenaAllocator\* alloc;

// Load scope attributes and push onto scope stack.

scope->SetIsDynamic(this->isDynamic);

if (this->isObject)

{

scope->SetIsObject();

}

scope->SetMustInstantiate(this->mustInstantiate);

if (parser)

{

scriptContext = parser->GetScriptContext();

alloc = parser->GetAllocator();

}

else

{

TRACE\_BYTECODE(L"\nRestore ScopeInfo: %s #symbols: %d %s\n",

funcInfo->name, symbolCount, isObject ? L"isObject" : L"");

Assert(!this->isCached || scope == funcInfo->GetBodyScope());

funcInfo->SetHasCachedScope(this->isCached);

byteCodeGenerator->PushScope(scope);

if (byteCodeGenerator->UseParserBindings())

{

// The scope is already populated, so we're done.

return;

}

scriptContext = byteCodeGenerator->GetScriptContext();

alloc = byteCodeGenerator->GetAllocator();

}

// Load scope symbols

// On first access to the scopeinfo, replace the ID's with PropertyRecord\*'s to save the dictionary lookup

// on later accesses. Replace them all before allocating Symbol's to prevent inconsistency on OOM.

if (!this->areNamesCached && !PHASE\_OFF1(Js::CacheScopeInfoNamesPhase))

{

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

{

PropertyId propertyId = GetSymbolId(i);

if (propertyId != 0) // There may be empty slots, e.g. "arguments" may have no slot

{

PropertyRecord const\* name = scriptContext->GetPropertyName(propertyId);

this->SetPropertyName(i, name);

}

}

this->areNamesCached = true;

}

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

{

PropertyRecord const\* name = nullptr;

if (this->areNamesCached)

{

name = this->GetPropertyName(i);

}

else

{

PropertyId propertyId = GetSymbolId(i);

if (propertyId != 0) // There may be empty slots, e.g. "arguments" may have no slot

{

name = scriptContext->GetPropertyName(propertyId);

}

}

if (name != nullptr)

{

SymbolType symbolType = GetSymbolType(i);

SymbolName symName(name->GetBuffer(), name->GetLength());

Symbol \*sym = Anew(alloc, Symbol, symName, nullptr, symbolType);

sym->SetScopeSlot(static\_cast<PropertyId>(i));

sym->SetIsBlockVar(GetIsBlockVariable(i));

if (GetHasFuncAssignment(i))

{

sym->RestoreHasFuncAssignment();

}

scope->AddNewSymbol(sym);

if (parser)

{

parser->RestorePidRefForSym(sym);

}

TRACE\_BYTECODE(L"%12s %d\n", sym->GetName().GetBuffer(), sym->GetScopeSlot());

}

}

this->scope = scope;

DebugOnly(scope->isRestored = true);

}

ScopeInfo::AutoCapturesAllScope::AutoCapturesAllScope(Scope\* scope, bool turnOn)

: scope(scope)

{

oldCapturesAll = scope->GetCapturesAll();

if (turnOn)

{

scope->SetCapturesAll(true);

}

}

ScopeInfo::AutoCapturesAllScope::~AutoCapturesAllScope()

{

scope->SetCapturesAll(oldCapturesAll);

}

} // 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.

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

#pragma once

namespace Js {

//

// ScopeInfo is used to persist Scope info of outer functions. When reparsing deferred nested

// functions, use persisted ScopeInfo to restore outer closures.

//

class ScopeInfo

{

struct MapSymbolData

{

ByteCodeGenerator\* byteCodeGenerator;

FuncInfo\* func;

};

struct SymbolInfo

{

union

{

PropertyId propertyId;

PropertyRecord const\* name;

};

SymbolType symbolType;

bool hasFuncAssignment;

bool isBlockVariable;

};

private:

FunctionBody \* const parent; // link to parent function body

ScopeInfo\* funcExprScopeInfo; // optional func expr scope info

ScopeInfo\* paramScopeInfo; // optional param scope info

BYTE isDynamic : 1; // isDynamic bit affects how deferredChild access global ref

BYTE isObject : 1; // isObject bit affects how deferredChild access closure symbols

BYTE mustInstantiate : 1; // the scope must be instantiated as an object/array

BYTE isCached : 1; // indicates that local vars and functions are cached across invocations

BYTE isGlobalEval : 1;

BYTE areNamesCached : 1;

Scope \*scope;

int scopeId;

int symbolCount; // symbol count in this scope

SymbolInfo symbols[]; // symbol PropertyIDs, index == sym.scopeSlot

private:

ScopeInfo(FunctionBody \* parent, int symbolCount)

: parent(parent), funcExprScopeInfo(nullptr), paramScopeInfo(nullptr), symbolCount(symbolCount), scope(nullptr), areNamesCached(false)

{

}

void SetFuncExprScopeInfo(ScopeInfo\* funcExprScopeInfo)

{

this->funcExprScopeInfo = funcExprScopeInfo;

}

void SetParamScopeInfo(ScopeInfo\* paramScopeInfo)

{

this->paramScopeInfo = paramScopeInfo;

}

void SetSymbolId(int i, PropertyId propertyId)

{

Assert(!areNamesCached);

Assert(i >= 0 && i < symbolCount);

symbols[i].propertyId = propertyId;

}

void SetSymbolType(int i, SymbolType symbolType)

{

Assert(!areNamesCached);

Assert(i >= 0 && i < symbolCount);

symbols[i].symbolType = symbolType;

}

void SetHasFuncAssignment(int i, bool has)

{

Assert(!areNamesCached);

Assert(i >= 0 && i < symbolCount);

symbols[i].hasFuncAssignment = has;

}

void SetIsBlockVariable(int i, bool is)

{

Assert(!areNamesCached);

Assert(i >= 0 && i < symbolCount);

symbols[i].isBlockVariable = is;

}

void SetPropertyName(int i, PropertyRecord const\* name)

{

Assert(!areNamesCached);

Assert(i >= 0 && i < symbolCount);

symbols[i].name = name;

}

PropertyId GetSymbolId(int i) const

{

Assert(!areNamesCached);

Assert(i >= 0 && i < symbolCount);

return symbols[i].propertyId;

}

SymbolType GetSymbolType(int i) const

{

Assert(i >= 0 && i < symbolCount);

return symbols[i].symbolType;

}

bool GetHasFuncAssignment(int i)

{

Assert(i >= 0 && i < symbolCount);

return symbols[i].hasFuncAssignment;

}

bool GetIsBlockVariable(int i)

{

Assert(i >= 0 && i < symbolCount);

return symbols[i].isBlockVariable;

}

PropertyRecord const\* GetPropertyName(int i)

{

Assert(areNamesCached);

Assert(i >= 0 && i < symbolCount);

return symbols[i].name;

}

void SaveSymbolInfo(Symbol\* sym, MapSymbolData\* mapSymbolData);

static ScopeInfo\* FromParent(FunctionBody\* parent);

static ScopeInfo\* FromScope(ByteCodeGenerator\* byteCodeGenerator, FunctionBody\* parent, Scope\* scope, ScriptContext \*scriptContext);

static void SaveParentScopeInfo(FuncInfo\* parentFunc, FuncInfo\* func);

static void SaveScopeInfo(ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* parentFunc, FuncInfo\* func);

public:

FunctionBody \* GetParent() const

{

return parent;

}

ScopeInfo\* GetParentScopeInfo() const

{

return parent->GetScopeInfo();

}

ScopeInfo\* GetFuncExprScopeInfo() const

{

return funcExprScopeInfo;

}

ScopeInfo\* GetParamScopeInfo() const

{

return paramScopeInfo;

}

void SetScope(Scope \*scope)

{

this->scope = scope;

}

Scope \* GetScope() const

{

return scope;

}

void SetScopeId(int id)

{

this->scopeId = id;

}

int GetScopeId() const

{

return scopeId;

}

int GetSymbolCount() const

{

return symbolCount;

}

bool IsGlobalEval() const

{

return isGlobalEval;

}

static void SaveScopeInfoForDeferParse(ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* parentFunc, FuncInfo\* func);

ScopeInfo \*CloneFor(ParseableFunctionInfo \*body);

void EnsurePidTracking(ScriptContext\* scriptContext);

void GetScopeInfo(Parser \*parser, ByteCodeGenerator\* byteCodeGenerator, FuncInfo\* funcInfo, Scope\* scope);

//

// Turn on capturesAll for a Scope temporarily. Restore old capturesAll when this object

// goes out of scope.

//

class AutoCapturesAllScope

{

private:

Scope\* scope;

bool oldCapturesAll;

public:

AutoCapturesAllScope(Scope\* scope, bool turnOn);

~AutoCapturesAllScope();

bool OldCapturesAll() const

{

return oldCapturesAll;

}

};

};

}

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

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

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

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

#include "RuntimeByteCodePch.h"

namespace Js

{

void StatementReader::Create(FunctionBody \* functionRead, uint startOffset /\* = 0 \*/)

{

Assert(functionRead);

StatementReader::Create(functionRead, startOffset, false);

}

void StatementReader::Create(FunctionBody\* functionRead, uint startOffset, bool useOriginalByteCode)

{

AssertMsg(functionRead != nullptr, "Must provide valid function to execute");

ByteBlock \* pblkByteCode = useOriginalByteCode ?

functionRead->GetOriginalByteCode() :

functionRead->GetByteCode();

AssertMsg(pblkByteCode != nullptr, "Must have valid byte-code to read");

m\_startLocation = pblkByteCode->GetBuffer();

const byte \* currentLocation = m\_startLocation + startOffset;

const bool isInDebugMode = functionRead->IsByteCodeDebugMode();

m\_statementMap = functionRead->GetStatementMapSpanSequence();

if (m\_statementMap == nullptr && isInDebugMode)

{

m\_fullstatementMap = functionRead->GetStatementMaps();

}

else

{

m\_fullstatementMap = nullptr;

}

if (m\_statementMap && m\_statementMap->Count())

{

m\_statementMap->Reset(m\_statementMapIter);

m\_statementIndex = 0;

m\_startOfStatement = true;

StatementData data;

if (!m\_statementMap->Seek(m\_statementIndex, data))

{

Assert(FALSE);

}

m\_nextStatementBoundary = m\_startLocation + data.bytecodeBegin;

// If we starting in the middle of the function (e.g., loop body), find out where the next statement is.

while (m\_nextStatementBoundary < currentLocation)

{

this->MoveNextStatementBoundary();

}

}

else if (m\_fullstatementMap && m\_fullstatementMap->Count())

{

m\_statementIndex = 0;

m\_startOfStatement = true;

FunctionBody::StatementMap \*nextMap = Js::FunctionBody::GetNextNonSubexpressionStatementMap(m\_fullstatementMap, m\_statementIndex);

if (!nextMap)

{

// set to a location that will never match

m\_nextStatementBoundary = currentLocation - 1;

}

else

{

m\_nextStatementBoundary = m\_startLocation + m\_fullstatementMap->Item(m\_statementIndex)->byteCodeSpan.begin;

// If we starting in the middle of the function (e.g., loop body), find out where the next statement is.

while (m\_nextStatementBoundary < currentLocation)

{

this->MoveNextStatementBoundary();

}

}

}

else

{

// set to a location that will never match

m\_nextStatementBoundary = currentLocation - 1;

}

}

inline uint32 StatementReader::MoveNextStatementBoundary()

{

StatementData data;

uint32 retStatement = Js::Constants::NoStatementIndex;

if (m\_startOfStatement)

{

m\_statementIndex++;

if (m\_statementMap && (uint32)m\_statementIndex < m\_statementMap->Count() && m\_statementMap->Item(m\_statementIndex, m\_statementMapIter, data))

{

// The end boundary is the last byte of the last instruction in the previous range.

// We want to track the beginning of the next instruction for AtStatementBoundary.

m\_nextStatementBoundary = m\_startLocation + data.bytecodeBegin;

// The next user statement is adjacent in the bytecode

retStatement = m\_statementIndex;

}

else if (m\_fullstatementMap && m\_statementIndex < m\_fullstatementMap->Count())

{

int nextInstrStart = m\_fullstatementMap->Item(m\_statementIndex - 1)->byteCodeSpan.end + 1;

m\_nextStatementBoundary = m\_startLocation + nextInstrStart;

Js::FunctionBody::GetNextNonSubexpressionStatementMap(m\_fullstatementMap, m\_statementIndex);

if (nextInstrStart == m\_fullstatementMap->Item(m\_statementIndex)->byteCodeSpan.begin)

{

retStatement = m\_statementIndex;

}

else

{

m\_startOfStatement = false;

}

}

else

{

m\_startOfStatement = false;

}

}

else

{

m\_startOfStatement = true;

if (m\_statementMap && (uint32)m\_statementIndex < m\_statementMap->Count() && m\_statementMap->Item(m\_statementIndex, m\_statementMapIter, data))

{

// Start a range of bytecode that maps to a user statement

m\_nextStatementBoundary = m\_startLocation + data.bytecodeBegin;

retStatement = m\_statementIndex;

}

else if (m\_fullstatementMap && m\_statementIndex < m\_fullstatementMap->Count())

{

FunctionBody::StatementMap \*nextMap = Js::FunctionBody::GetNextNonSubexpressionStatementMap(m\_fullstatementMap, m\_statementIndex);

if (!nextMap)

{

// set to a location that will never match

m\_nextStatementBoundary = m\_startLocation - 1;

}

else

{

// Start a range of bytecode that maps to a user statement

m\_nextStatementBoundary = m\_startLocation + m\_fullstatementMap->Item(m\_statementIndex)->byteCodeSpan.begin;

retStatement = m\_statementIndex;

}

}

else

{

// The remaining bytecode instructions do not map to a user statement, set a statementBoundary that cannot match

m\_nextStatementBoundary = m\_startLocation - 1;

}

}

return retStatement;

}

} // 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.

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

#pragma once

namespace Js

{

struct StatementReader

{

private:

const byte\* m\_startLocation;

SmallSpanSequence\* m\_statementMap;

SmallSpanSequenceIter m\_statementMapIter;

FunctionBody::StatementMapList\* m\_fullstatementMap;

const byte\* m\_nextStatementBoundary;

int m\_statementIndex;

bool m\_startOfStatement;

public:

void Create(FunctionBody\* functionRead, uint startOffset = 0);

void Create(FunctionBody\* functionRead, uint startOffset, bool useOriginalByteCode);

inline bool AtStatementBoundary(ByteCodeReader \* reader) { return m\_nextStatementBoundary == reader->GetIP(); }

inline uint32 MoveNextStatementBoundary();

inline uint32 GetStatementIndex() const { return m\_statementIndex; }

};

} // 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 "RuntimeByteCodePch.h"

#if DBG\_DUMP

static const wchar\_t \* const SymbolTypeNames[] = { L"Function", L"Variable", L"MemberName", L"Formal", L"Unknown" };

#endif

bool Symbol::GetIsArguments() const

{

return decl != nullptr && (decl->grfpn & PNodeFlags::fpnArguments);

}

Js::PropertyId Symbol::EnsurePosition(ByteCodeGenerator\* byteCodeGenerator)

{

// Guarantee that a symbol's name has a property ID mapping.

if (this->position == Js::Constants::NoProperty)

{

this->position = this->EnsurePositionNoCheck(byteCodeGenerator->TopFuncInfo());

}

return this->position;

}

Js::PropertyId Symbol::EnsurePosition(FuncInfo \*funcInfo)

{

// Guarantee that a symbol's name has a property ID mapping.

if (this->position == Js::Constants::NoProperty)

{

this->position = this->EnsurePositionNoCheck(funcInfo);

}

return this->position;

}

Js::PropertyId Symbol::EnsurePositionNoCheck(FuncInfo \*funcInfo)

{

return funcInfo->byteCodeFunction->GetOrAddPropertyIdTracked(this->GetName());

}

void Symbol::SaveToPropIdArray(Symbol \*sym, Js::PropertyIdArray \*propIds, ByteCodeGenerator \*byteCodeGenerator, Js::PropertyId \*pFirstSlot /\* = null \*/)

{

if (sym)

{

Js::PropertyId slot = sym->scopeSlot;

if (slot != Js::Constants::NoProperty)

{

Assert((uint32)slot < propIds->count);

propIds->elements[slot] = sym->EnsurePosition(byteCodeGenerator);

if (pFirstSlot && !sym->GetIsArguments())

{

if (\*pFirstSlot == Js::Constants::NoProperty ||

\*pFirstSlot > slot)

{

\*pFirstSlot = slot;

}

}

}

}

}

bool Symbol::NeedsSlotAlloc(FuncInfo \*funcInfo)

{

return IsInSlot(funcInfo, true);

}

bool Symbol::IsInSlot(FuncInfo \*funcInfo, bool ensureSlotAlloc)

{

if (this->GetIsGlobal())

{

return false;

}

if (funcInfo->GetHasHeapArguments() && this->GetIsFormal() && ByteCodeGenerator::NeedScopeObjectForArguments(funcInfo, funcInfo->root))

{

// Rest is a special case - it will be in a register.

if (funcInfo->root->sxFnc.pnodeRest != this->decl)

{

return true;

}

}

if (this->GetIsGlobalCatch())

{

return true;

}

if (this->scope->GetCapturesAll())

{

return true;

}

return this->GetHasNonLocalReference() && (ensureSlotAlloc || this->GetIsCommittedToSlot());

}

bool Symbol::GetIsCommittedToSlot() const

{

return isCommittedToSlot || this->scope->GetFunc()->GetCallsEval() || this->scope->GetFunc()->GetChildCallsEval();

}

Js::PropertyId Symbol::EnsureScopeSlot(FuncInfo \*funcInfo)

{

if (this->NeedsSlotAlloc(funcInfo) && this->scopeSlot == Js::Constants::NoProperty)

{

this->scopeSlot = this->scope->AddScopeSlot();

}

return this->scopeSlot;

}

void Symbol::SetHasNonLocalReference(bool b, ByteCodeGenerator \*byteCodeGenerator)

{

this->hasNonLocalReference = b;

// The symbol's home function will tell us which child function we're currently processing.

// This is the one that captures the symbol, from the declaring function's perspective.

// So based on that information, note either that, (a.) the symbol is committed to the heap from its

// inception, (b.) the symbol must be committed when the capturing function is instantiated.

FuncInfo \*funcHome = this->scope->GetFunc();

FuncInfo \*funcChild = funcHome->GetCurrentChildFunction();

// If this is not a local property, or not all its references can be tracked, or

// it's not scoped to the function, or we're in debug mode, disable the delayed capture optimization.

if (funcHome->IsGlobalFunction() ||

funcHome->GetCallsEval() ||

funcHome->GetChildCallsEval() ||

funcChild == nullptr ||

this->GetScope() != funcHome->GetBodyScope() ||

byteCodeGenerator->IsInDebugMode() ||

PHASE\_OFF(Js::DelayCapturePhase, funcHome->byteCodeFunction))

{

this->SetIsCommittedToSlot();

}

if (this->isCommittedToSlot)

{

return;

}

AnalysisAssert(funcChild);

ParseNode \*pnodeChild = funcChild->root;

Assert(pnodeChild && pnodeChild->nop == knopFncDecl);

if (pnodeChild->sxFnc.IsDeclaration())

{

// The capturing function is a declaration but may still be limited to an inner scope.

Scope \*scopeChild = funcHome->GetCurrentChildScope();

if (scopeChild == this->scope || scopeChild->GetScopeType() == ScopeType\_FunctionBody)

{

// The symbol is captured on entry to the scope in which it's declared.

// (Check the scope type separately so that we get the special parameter list and

// named function expression cases as well.)

this->SetIsCommittedToSlot();

return;

}

}

// There is a chance we can limit the region in which the symbol lives on the heap.

// Note which function captures the symbol.

funcChild->AddCapturedSym(this);

}

void Symbol::SetHasMaybeEscapedUse(ByteCodeGenerator \* byteCodeGenerator)

{

Assert(!this->GetIsMember());

if (!hasMaybeEscapedUse)

{

SetHasMaybeEscapedUseInternal(byteCodeGenerator);

}

}

void Symbol::SetHasMaybeEscapedUseInternal(ByteCodeGenerator \* byteCodeGenerator)

{

Assert(!hasMaybeEscapedUse);

Assert(!this->GetIsFormal());

hasMaybeEscapedUse = true;

if (PHASE\_TESTTRACE(Js::StackFuncPhase, byteCodeGenerator->TopFuncInfo()->byteCodeFunction))

{

Output::Print(L"HasMaybeEscapedUse: %s\n", this->GetName().GetBuffer());

Output::Flush();

}

if (this->GetHasFuncAssignment())

{

this->GetScope()->GetFunc()->SetHasMaybeEscapedNestedFunc(

DebugOnly(this->symbolType == STFunction ? L"MaybeEscapedUseFuncDecl" : L"MaybeEscapedUse"));

}

}

void Symbol::SetHasFuncAssignment(ByteCodeGenerator \* byteCodeGenerator)

{

Assert(!this->GetIsMember());

if (!hasFuncAssignment)

{

SetHasFuncAssignmentInternal(byteCodeGenerator);

}

}

void Symbol::SetHasFuncAssignmentInternal(ByteCodeGenerator \* byteCodeGenerator)

{

Assert(!hasFuncAssignment);

hasFuncAssignment = true;

FuncInfo \* top = byteCodeGenerator->TopFuncInfo();

if (PHASE\_TESTTRACE(Js::StackFuncPhase, top->byteCodeFunction))

{

Output::Print(L"HasFuncAssignment: %s\n", this->GetName().GetBuffer());

Output::Flush();

}

if (this->GetHasMaybeEscapedUse() || this->GetScope()->GetIsObject())

{

byteCodeGenerator->TopFuncInfo()->SetHasMaybeEscapedNestedFunc(DebugOnly(

this->GetIsFormal() ? L"FormalAssignment" :

this->GetScope()->GetIsObject() ? L"ObjectScopeAssignment" :

L"MaybeEscapedUse"));

}

}

void Symbol::RestoreHasFuncAssignment()

{

Assert(hasFuncAssignment == (this->symbolType == STFunction));

Assert(this->GetIsFormal() || !this->GetHasMaybeEscapedUse());

hasFuncAssignment = true;

if (PHASE\_TESTTRACE1(Js::StackFuncPhase))

{

Output::Print(L"RestoreHasFuncAssignment: %s\n", this->GetName().GetBuffer());

Output::Flush();

}

}

Symbol \* Symbol::GetFuncScopeVarSym() const

{

if (!this->GetIsBlockVar())

{

return nullptr;

}

FuncInfo \* parentFuncInfo = this->GetScope()->GetFunc();

if (parentFuncInfo->GetIsStrictMode())

{

return nullptr;

}

Symbol \*fncScopeSym = parentFuncInfo->GetBodyScope()->FindLocalSymbol(this->GetName());

if (fncScopeSym == nullptr && parentFuncInfo->GetParamScope() != nullptr)

{

// We couldn't find the sym in the body scope, try finding it in the parameter scope.

fncScopeSym = parentFuncInfo->GetParamScope()->FindLocalSymbol(this->GetName());

}

Assert(fncScopeSym);

// Parser should have added a fake var decl node for block scoped functions in non-strict mode

// IsBlockVar() indicates a user let declared variable at function scope which

// shadows the function's var binding, thus only emit the var binding init if

// we do not have a block var symbol.

if (!fncScopeSym || fncScopeSym->GetIsBlockVar())

{

return nullptr;

}

return fncScopeSym;

}

#if DBG\_DUMP

const wchar\_t \* Symbol::GetSymbolTypeName()

{

return SymbolTypeNames[symbolType];

}

#endif

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

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

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

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

enum SymbolType : byte

{

STFunction,

STVariable,

STMemberName,

STFormal,

STUnknown

};

typedef JsUtil::CharacterBuffer<WCHAR> SymbolName;

class Symbol

{

private:

const SymbolName name;

IdentPtr pid;

ParseNode \*decl;

Scope \*scope; // scope defining this symbol

Js::PropertyId position; // argument position in function declaration

Js::RegSlot location; // register in which the symbol resides

Js::PropertyId scopeSlot;

Symbol \*next;

SymbolType symbolType;

BYTE defCount;

BYTE needDeclaration : 1;

BYTE isBlockVar : 1;

BYTE isGlobal : 1;

BYTE isEval : 1;

BYTE hasNonLocalReference : 1; // if true, then this symbol needs to be heap-allocated

BYTE funcExpr : 1; // if true, then this symbol is allocated on it's on activation object

BYTE isCatch : 1; // if true then this a catch identifier

BYTE hasInit : 1;

BYTE isUsed : 1;

BYTE isGlobalCatch : 1;

BYTE isCommittedToSlot : 1;

BYTE hasNonCommittedReference : 1;

BYTE hasVisitedCapturingFunc : 1;

BYTE isTrackedForDebugger : 1; // Whether the sym is tracked for debugger scope. This is fine because a sym can only be added to (not more than) one scope.

// These are get and set a lot, don't put it in bit fields, we are exceeding the number of bits anyway

bool hasFuncAssignment;

bool hasMaybeEscapedUse;

bool isNonSimpleParameter;

AssignmentState assignmentState;

public:

Symbol(SymbolName const& name, ParseNode \*decl, SymbolType symbolType) :

name(name),

decl(decl),

next(nullptr),

location(Js::Constants::NoRegister),

needDeclaration(false),

isBlockVar(false),

isGlobal(false),

hasNonLocalReference(false),

funcExpr(false),

isCatch(false),

hasInit(false),

isUsed(false),

defCount(0),

position(Js::Constants::NoProperty),

scopeSlot(Js::Constants::NoProperty),

isGlobalCatch(false),

isCommittedToSlot(false),

hasNonCommittedReference(false),

hasVisitedCapturingFunc(false),

isTrackedForDebugger(false),

isNonSimpleParameter(false),

assignmentState(NotAssigned)

{

SetSymbolType(symbolType);

// Set it so we don't have to check it explicitly

isEval = MatchName(L"eval", 4);

if (PHASE\_TESTTRACE1(Js::StackFuncPhase) && hasFuncAssignment)

{

Output::Print(L"HasFuncDecl: %s\n", this->GetName().GetBuffer());

Output::Flush();

}

}

bool MatchName(const wchar\_t \*key, int length)

{

return name == SymbolName(key, length);

}

void SetScope(Scope \*scope)

{

this->scope = scope;

}

Scope \* GetScope() const { return scope; }

void SetDecl(ParseNode \*pnodeDecl) { decl = pnodeDecl; }

ParseNode\* GetDecl() const { return decl; }

void SetScopeSlot(Js::PropertyId slot)

{

this->scopeSlot = slot;

}

Symbol \*GetNext() const

{

return next;

}

void SetNext(Symbol \*sym)

{

next = sym;

}

void SetIsGlobal(bool b)

{

isGlobal = b;

}

void SetHasNonLocalReference(bool b, ByteCodeGenerator \*byteCodeGenerator);

bool GetHasNonLocalReference() const

{

return hasNonLocalReference;

}

void SetFuncExpr(bool b)

{

funcExpr = b;

}

void SetIsBlockVar(bool is)

{

isBlockVar = is;

}

bool GetIsBlockVar() const

{

return isBlockVar;

}

void SetIsGlobalCatch(bool is)

{

isGlobalCatch = is;

}

bool GetIsGlobalCatch() const

{

return isGlobalCatch;

}

void SetIsCommittedToSlot()

{

this->isCommittedToSlot = true;

}

bool GetIsCommittedToSlot() const;

void SetHasVisitedCapturingFunc()

{

this->hasVisitedCapturingFunc = true;

}

bool HasVisitedCapturingFunc() const

{

return hasVisitedCapturingFunc;

}

void SetHasNonCommittedReference(bool has)

{

this->hasNonCommittedReference = has;

}

bool GetHasNonCommittedReference() const

{

return hasNonCommittedReference;

}

void SetIsTrackedForDebugger(bool is)

{

isTrackedForDebugger = is;

}

bool GetIsTrackedForDebugger() const

{

return isTrackedForDebugger;

}

void SetNeedDeclaration(bool need)

{

needDeclaration = need;

}

bool GetNeedDeclaration() const

{

return needDeclaration;

}

bool GetFuncExpr() const

{

return funcExpr;

}

bool GetIsGlobal() const

{

return isGlobal;

}

bool GetIsMember() const

{

return symbolType == STMemberName;

}

bool GetIsFormal() const

{

return symbolType == STFormal;

}

bool GetIsEval() const

{

return isEval;

}

bool GetIsCatch() const

{

return isCatch;

}

void SetIsCatch(bool b)

{

isCatch = b;

}

bool GetHasInit() const

{

return hasInit;

}

void RecordDef()

{

defCount++;

}

bool SingleDef() const

{

return defCount == 1;

}

void SetHasInit(bool has)

{

hasInit = has;

}

bool GetIsUsed() const

{

return isUsed;

}

void SetIsUsed(bool is)

{

isUsed = is;

}

void PromoteAssignmentState()

{

if (assignmentState == NotAssigned)

{

assignmentState = AssignedOnce;

}

else if (assignmentState == AssignedOnce)

{

assignmentState = AssignedMultipleTimes;

}

}

bool IsAssignedOnce()

{

return assignmentState == AssignedOnce;

}

// For stack nested function escape analysis

bool GetHasMaybeEscapedUse() const { return hasMaybeEscapedUse; }

void SetHasMaybeEscapedUse(ByteCodeGenerator \* byteCodeGenerator);

bool GetHasFuncAssignment() const { return hasFuncAssignment; }

void SetHasFuncAssignment(ByteCodeGenerator \* byteCodeGenerator);

void RestoreHasFuncAssignment();

bool GetIsNonSimpleParameter() const

{

return isNonSimpleParameter;

}

void SetIsNonSimpleParameter(bool is)

{

isNonSimpleParameter = is;

}

bool GetIsArguments() const;

void SetPosition(Js::PropertyId pos)

{

position = pos;

}

Js::PropertyId GetPosition()

{

return position;

}

Js::PropertyId EnsurePosition(ByteCodeGenerator\* byteCodeGenerator);

Js::PropertyId EnsurePosition(FuncInfo \*funcInfo);

Js::PropertyId EnsurePositionNoCheck(FuncInfo \*funcInfo);

void SetLocation(Js::RegSlot location)

{

this->location = location;

}

Js::RegSlot GetLocation()

{

return location;

}

Js::PropertyId GetScopeSlot() const { return scopeSlot; }

bool HasScopeSlot() const { return scopeSlot != Js::Constants::NoProperty; }

SymbolType GetSymbolType()

{

return symbolType;

}

void SetSymbolType(SymbolType symbolType)

{

this->symbolType = symbolType;

this->hasMaybeEscapedUse = GetIsFormal();

this->hasFuncAssignment = (symbolType == STFunction);

}

#if DBG\_DUMP

const wchar\_t \*GetSymbolTypeName();

#endif

const JsUtil::CharacterBuffer<WCHAR>& GetName() const

{

return this->name;

}

Js::PropertyId EnsureScopeSlot(FuncInfo \*funcInfo);

bool IsInSlot(FuncInfo \*funcInfo, bool ensureSlotAlloc = false);

bool NeedsSlotAlloc(FuncInfo \*funcInfo);

static void SaveToPropIdArray(Symbol \*sym, Js::PropertyIdArray \*propIds, ByteCodeGenerator \*byteCodeGenerator, Js::PropertyId \*pFirstSlot = nullptr);

Symbol \* GetFuncScopeVarSym() const;

void SetPid(IdentPtr pid)

{

this->pid = pid;

}

IdentPtr GetPid() const

{

return pid;

}

private:

void SetHasMaybeEscapedUseInternal(ByteCodeGenerator \* byteCodeGenerator);

void SetHasFuncAssignmentInternal(ByteCodeGenerator \* byteCodeGenerator);

};

// specialize toKey to use the name in the symbol as the key

template <>

SymbolName JsUtil::ValueToKey<SymbolName, Symbol \*>::ToKey(Symbol \* const& sym)

{

return sym->GetName();

}

typedef JsUtil::BaseHashSet<Symbol \*, ArenaAllocator, PrimeSizePolicy, SymbolName, DefaultComparer, JsUtil::HashedEntry> SymbolTable;

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

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

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

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

#pragma once

namespace Js

{

struct VarArrayVarCount

{

Var count;

Var elements[];

VarArrayVarCount(Var count) : count(count)

{

}

void SetCount(uint count);

size\_t GetDataSize() const;

};

};