Source code

In this file, we listed the most important source code related to the final version of the generic JavaScript-to-WebAssembly wrapper function written by us:

- the generic wrapper function,
- GC support,
- tests.

Note: all the source code below is from the V8 repository.

Source code of the generic wrapper

```
// Copyright 2012 the V8 project authors. All rights reserved.
// Use of this source code is governed by a BSD-style license that can be
// found in the LICENSE file.

namespace {
// Helper functions for the GenericJSToWasmWrapper.
void PrepareForBuiltinCall(MacroAssembler* masm, MemOperand GCScanSlotPlace,
                           const int GCScanSlotCount, Register current_param,
                           Register param_limit,
                           Register current_int_param_slot,
                           Register current_float_param_slot,
                           Register valuetypes_array_ptr,
                           Register wasm_instance, Register function_data) {
  // Pushes and puts the values in order onto the stack before builtin calls for
  // the GenericJSToWasmWrapper.
  __ movq(GCScanSlotPlace, Immediate(GCScanSlotCount));
  __ pushq(current_param);
  __ pushq(param_limit);
  __ pushq(current_int_param_slot);
  __ pushq(current_float_param_slot);
  __ pushq(valuetypes_array_ptr);
  __ pushq(wasm_instance);
  __ pushq(function_data);
  // We had to prepare the parameters for the Call: we have to put the context
  // into rsi.
  __ LoadAnyTaggedField(
      rsi,
      MemOperand(wasm_instance, wasm::ObjectAccess::ToTagged(
                                    WasmInstanceObject::kNativeContextOffset)));
}

void RestoreAfterBuiltinCall(MacroAssembler* masm, Register function_data,
                             Register wasm_instance,
                             Register valuetypes_array_ptr,
                             Register current_float_param_slot,
                             Register current_int_param_slot,
                             Register param_limit, Register current_param) {
  // Pop and load values from the stack in order into the registers after
  // builtin calls for the GenericJSToWasmWrapper.
  __ popq(function_data);
  __ popq(wasm_instance);
  __ popq(valuetypes_array_ptr);
  __ popq(current_float_param_slot);
  __ popq(current_int_param_slot);
  __ popq(param_limit);
  __ popq(current_param);
}
}  // namespace

void Builtins::Generate_GenericJSToWasmWrapper(MacroAssembler* masm) {
  // Set up the stackframe.
  __ EnterFrame(StackFrame::JS_TO_WASM);

  // -------------------------------------------
  // Load the Wasm exported function data and the Wasm instance.
```

```
// -----------------------------------------
Register closure = rdi;
Register shared_function_info = closure;
__ LoadAnyTaggedField(
    shared_function_info,
    MemOperand(
        closure,
        wasm::ObjectAccess::SharedFunctionInfoOffsetInTaggedJSFunction()));
closure = no_reg;
Register function_data = shared_function_info;
__ LoadAnyTaggedField(
    function_data,
    MemOperand(shared_function_info,
               SharedFunctionInfo::kFunctionDataOffset - kHeapObjectTag));
shared_function_info = no_reg;

Register wasm_instance = rsi;
__ LoadAnyTaggedField(
    wasm_instance,
    MemOperand(function_data,
               WasmExportedFunctionData::kInstanceOffset - kHeapObjectTag));

// -----------------------------------------
// Load values from the signature.
// -----------------------------------------
Register foreign_signature = r11;
__ LoadAnyTaggedField(
    foreign_signature,
    MemOperand(function_data,
               WasmExportedFunctionData::kSignatureOffset - kHeapObjectTag));
Register signature = foreign_signature;
__ movq(signature,
        MemOperand(foreign_signature, wasm::ObjectAccess::ToTagged(
                                          Foreign::kForeignAddressOffset)));
foreign_signature = no_reg;
Register return_count = r8;
__ movq(return_count,
        MemOperand(signature, wasm::FunctionSig::kReturnCountOffset));
Register param_count = rcx;
__ movq(param_count,
        MemOperand(signature, wasm::FunctionSig::kParameterCountOffset));
Register valuetypes_array_ptr = signature;
__ movq(valuetypes_array_ptr,
        MemOperand(signature, wasm::FunctionSig::kRepsOffset));
signature = no_reg;

// -----------------------------------------
// Set up the stack.
// -----------------------------------------
// We store values on the stack to restore them after function calls.
// We cannot push values onto the stack right before the wasm call. The wasm
// function expects the parameters, that didn't fit into the registers, on the
// top of the stack.
// We will have to save a value indicating the GC the number
// of values on the top of the stack that have to be scanned before calling
// the Wasm function.
constexpr int kFrameMarkerOffset = -kSystemPointerSize;
constexpr int kGCScanSlotCountOffset =
    kFrameMarkerOffset - kSystemPointerSize;
constexpr int kParamCountOffset = kGCScanSlotCountOffset - kSystemPointerSize;
constexpr int kReturnCountOffset = kParamCountOffset - kSystemPointerSize;
constexpr int kValueTypesArrayStartOffset =
    kReturnCountOffset - kSystemPointerSize;
// We set and use this slot only when moving parameters into the parameter
// registers (so no GC scan is needed).
constexpr int kFunctionDataOffset =
    kValueTypesArrayStartOffset - kSystemPointerSize;
constexpr int kLastSpillOffset = kFunctionDataOffset;
constexpr int kNumSpillSlots = 5;
__ subq(rsp, Immediate(kNumSpillSlots * kSystemPointerSize));
```

```
__ movq(MemOperand(rbp, kParamCountOffset), param_count);
__ movq(MemOperand(rbp, kReturnCountOffset), return_count);
__ movq(MemOperand(rbp, kValueTypesArrayStartOffset), valuetypes_array_ptr);

// -------------------------------------------
// Parameter handling.
// -------------------------------------------
Label prepare_for_wasm_call;
__ cmpl(param_count, Immediate(0));

// IF we have 0 params: jump through parameter handling.
__ j(equal, &prepare_for_wasm_call);

// ELSE:
// Make sure we have the same number of arguments in order to be able to load
// the arguments using static offsets below.
__ cmpl(kJavaScriptCallArgCountRegister, param_count);
__ Check(equal, AbortReason::kInvalidNumberOfJsArgs);

// -------------------------------------------
// Create 2 sections for integer and float params.
// -------------------------------------------
// We will create 2 sections on the stack for the evaluated parameters:
// Integer and Float section, both with parameter count size. We will place
// the parameters into these sections depending on their valuetype. This way
// we can easily fill the general purpose and floating point parameter
// registers and place the remaining parameters onto the stack in proper order
// for the Wasm function. These remaining params are the final stack
// parameters for the call to WebAssembly. Example of the stack layout after
// processing 2 int and 1 float parameters when param_count is 4.
//   +-----------------+
//   |      rbp        |
//   |-----------------|-------------------------------
//   |                 |   Slots we defined
//   |   Saved values  |    when setting up
//   |                 |     the stack
//   |                 |
//   +-Integer section-+--- <--- start_int_section ----
//   |  1st int param  |
//   |- - - - - - - - -|
//   |  2nd int param  |
//   |- - - - - - - - -|  <----- current_int_param_slot
//   |                 |       (points to the stackslot
//   |- - - - - - - - -|  where the next int param should be placed)
//   |                 |
//   +--Float section--+--- <--- start_float_section --
//   | 1st float param |
//   |- - - - - - - - -|  <----  current_float_param_slot
//   |                 |       (points to the stackslot
//   |- - - - - - - - -|  where the next float param should be placed)
//   |                 |
//   |- - - - - - - - -|
//   |                 |
//   +---Final stack---+-------------------------------
//   +-parameters for--+-------------------------------
//   +-the Wasm call---+-------------------------------
//   |      . . .      |

constexpr int kIntegerSectionStartOffset =
    kLastSpillOffset - kSystemPointerSize;
// For Integer section.
// Set the current_int_param_slot to point to the start of the section.
Register current_int_param_slot = r14;
__ leaq(current_int_param_slot, MemOperand(rsp, -kSystemPointerSize));
Register params_size = param_count;
param_count = no_reg;
__ shlq(params_size, Immediate(kSystemPointerSizeLog2));
__ subq(rsp, params_size);

// For Float section.
```

```
  // Set the current_float_param_slot to point to the start of the section.
  Register current_float_param_slot = r15;
  __ leaq(current_float_param_slot, MemOperand(rsp, -kSystemPointerSize));
  __ subq(rsp, params_size);
  params_size = no_reg;
  param_count = rcx;
  __ movq(param_count, MemOperand(rbp, kParamCountOffset));

  // -----------------------------------------
  // Set up for the param evaluation loop.
  // -----------------------------------------
  // We will loop through the params starting with the 1st param.
  // The order of processing the params is important. We have to evaluate them
  // in an increasing order.
  //      Not reversed                Reversed
  //   +-----------------+------+-----------------+---------------
  //   |    receiver     |      |     param n     |
  //   |- - - - - - - - -|      |- - - - - - - - -|
  //   |      param 1    |      |    param n-1    |   Caller
  //   |       ...       |      |       ...       | frame slots
  //   |    param n-1    |      |     param 1     |
  //   |- - - - - - - - -|      |- - - - - - - - -|
  //   |     param n     |      |    receiver     |
  //  -+-----------------+------+-----------------+---------------
  //   |  return addr    |      |  return addr    |
  //   |- - - - - - - - -|<-FP->|- - - - - - - - -|
  //   |      rbp        |      |      rbp        |   Spill slots
  //   |- - - - - - - - -|      |- - - - - - - - -|
  //
  // [rbp + current_param] gives us the parameter we are processing.
  // We iterate through half-open interval <1st param, [rbp + param_limit]).

  Register current_param = rbx;
  Register param_limit = rdx;
#ifdef V8_REVERSE_JSARGS
  constexpr int kReceiverOnStackSize = kSystemPointerSize;
  __ movq(current_param,
          Immediate(kFPOnStackSize + kPCOnStackSize + kReceiverOnStackSize));
  __ movq(param_limit, param_count);
  __ shlq(param_limit, Immediate(kSystemPointerSizeLog2));
  __ addq(param_limit,
          Immediate(kFPOnStackSize + kPCOnStackSize + kReceiverOnStackSize));
  const int increment = kSystemPointerSize;
#else
  __ movq(current_param, param_count);
  __ shlq(current_param, Immediate(kSystemPointerSizeLog2));
  __ addq(current_param, Immediate(kFPOnStackSize));
  __ movq(param_limit, Immediate(kFPOnStackSize));
  const int increment = -kSystemPointerSize;
#endif
  Register param = rax;
  // We have to check the types of the params. The ValueType array contains
  // first the return then the param types.
  constexpr int kValueTypeSize = sizeof(wasm::ValueType);
  STATIC_ASSERT(kValueTypeSize == 4);
  const int32_t kValueTypeSizeLog2 = log2(kValueTypeSize);
  // Set the ValueType array pointer to point to the first parameter.
  Register returns_size = return_count;
  return_count = no_reg;
  __ shlq(returns_size, Immediate(kValueTypeSizeLog2));
  __ addq(valuetypes_array_ptr, returns_size);
  returns_size = no_reg;
  Register valuetype = r12;

  // -----------------------------------------
  // Param evaluation loop.
  // -----------------------------------------
  Label loop_through_params;
  __ bind(&loop_through_params);

```

```
__ movq(param, MemOperand(rbp, current_param, times_1, 0));
__ movl(valuetype,
        Operand(valuetypes_array_ptr, wasm::ValueType::bit_field_offset()));

// -------------------------------------------
// Param conversion.
// -------------------------------------------
// If param is a Smi we can easily convert it. Otherwise we'll call a builtin
// for conversion.
Label convert_param;
__ cmpq(valuetype, Immediate(wasm::kWasmI32.raw_bit_field()));
__ j(not_equal, &convert_param);
__ JumpIfNotSmi(param, &convert_param);
// Change the paramfrom Smi to int32.
__ SmiUntag(param);
// Zero extend.
__ movl(param, param);
// Place the param into the proper slot in Integer section.
__ movq(MemOperand(current_int_param_slot, 0), param);
__ subq(current_int_param_slot, Immediate(kSystemPointerSize));

// -------------------------------------------
// Param conversion done.
// -------------------------------------------
Label param_conversion_done;
__ bind(&param_conversion_done);

__ addq(current_param, Immediate(increment));
__ addq(valuetypes_array_ptr, Immediate(kValueTypeSize));

__ cmpq(current_param, param_limit);
__ j(not_equal, &loop_through_params);

// -------------------------------------------
// Move the parameters into the proper param registers.
// -------------------------------------------
// The Wasm function expects that the params can be popped from the top of the
// stack in an increasing order.
// We can always move the values on the beginning of the sections into the GP
// or FP parameter registers. If the parameter count is less than the number
// of parameter registers, we may move values into the registers that are not
// in the section.
// ----------- S t a t e -------------
//  -- r8  : start_int_section
//  -- rdi : start_float_section
//  -- r14 : current_int_param_slot
//  -- r15 : current_float_param_slot
//  -- r11 : valuetypes_array_ptr
//  -- r12 : valuetype
//  -- rsi : wasm_instance
//  -- GpParamRegisters = rax, rdx, rcx, rbx, r9
// -----------------------------------

Register temp_params_size = rax;
__ movq(temp_params_size, MemOperand(rbp, kParamCountOffset));
__ shlq(temp_params_size, Immediate(kSystemPointerSizeLog2));
// We want to use the register of the function_data = rdi.
__ movq(MemOperand(rbp, kFunctionDataOffset), function_data);
Register start_float_section = function_data;
function_data = no_reg;
__ movq(start_float_section, rbp);
__ addq(start_float_section, Immediate(kIntegerSectionStartOffset));
__ subq(start_float_section, temp_params_size);
temp_params_size = no_reg;
// Fill the FP param registers.
__ Movsd(xmm1, MemOperand(start_float_section, 0));
__ Movsd(xmm2, MemOperand(start_float_section, -kSystemPointerSize));
__ Movsd(xmm3, MemOperand(start_float_section, -2 * kSystemPointerSize));
__ Movsd(xmm4, MemOperand(start_float_section, -3 * kSystemPointerSize));
__ Movsd(xmm5, MemOperand(start_float_section, -4 * kSystemPointerSize));
```

```
__ Movsd(xmm6, MemOperand(start_float_section, -5 * kSystemPointerSize));
// We want the start to point to the last properly placed param.
__ subq(start_float_section, Immediate(5 * kSystemPointerSize));

Register start_int_section = r8;
__ movq(start_int_section, rbp);
__ addq(start_int_section, Immediate(kIntegerSectionStartOffset));
// Fill the GP param registers.
__ movq(rax, MemOperand(start_int_section, 0));
__ movq(rdx, MemOperand(start_int_section, -kSystemPointerSize));
__ movq(rcx, MemOperand(start_int_section, -2 * kSystemPointerSize));
__ movq(rbx, MemOperand(start_int_section, -3 * kSystemPointerSize));
__ movq(r9, MemOperand(start_int_section, -4 * kSystemPointerSize));
// We want the start to point to the last properly placed param.
__ subq(start_int_section, Immediate(4 * kSystemPointerSize));

// -------------------------------------------
// Place the final stack parameters to the proper place.
// -------------------------------------------
// We want the current_param_slot (insertion) pointers to point at the last
// param of the section instead of the next free slot.
__ addq(current_int_param_slot, Immediate(kSystemPointerSize));
__ addq(current_float_param_slot, Immediate(kSystemPointerSize));

// -------------------------------------------
// Final stack parameters loop.
// -------------------------------------------
// The parameters that didn't fit into the registers should be placed on the
// top of the stack contiguously. The interval of parameters between the
// start_section and the current_param_slot pointers define the remaining
// parameters of the section.
// We can iterate through the valuetypes array to decide from which section we
// need to push the parameter onto the top of the stack. By iterating in a
// reversed order we can easily pick the last parameter of the proper section.
// The parameter of the section is pushed on the top of the stack only if the
// interval of remaining params is not empty. This way we ensure that only
// params that didn't fit into param registers are pushed again.

Label loop_through_valuetypes;
__ bind(&loop_through_valuetypes);

// We iterated through the valuetypes array, we are one field over the end in
// the beginning. Also, we have to decrement it in each iteration.
__ subq(valuetypes_array_ptr, Immediate(kValueTypeSize));

// Check if there are still remaining integer params.
Label continue_loop;
__ cmpq(start_int_section, current_int_param_slot);
// If there are remaining integer params.
__ j(greater, &continue_loop);

// Check if there are still remaining float params.
__ cmpq(start_float_section, current_float_param_slot);
// If there aren't any params remaining.
Label params_done;
__ j(less_equal, &params_done);

__ bind(&continue_loop);
__ movl(valuetype,
        Operand(valuetypes_array_ptr, wasm::ValueType::bit_field_offset()));
Label place_integer_param;
Label place_float_param;
__ cmpq(valuetype, Immediate(wasm::kWasmI32.raw_bit_field()));
__ j(equal, &place_integer_param);

__ cmpq(valuetype, Immediate(wasm::kWasmI64.raw_bit_field()));
__ j(equal, &place_integer_param);

__ cmpq(valuetype, Immediate(wasm::kWasmF32.raw_bit_field()));
__ j(equal, &place_float_param);
```

```
__ cmpq(valuetype, Immediate(wasm::kWasmF64.raw_bit_field()));
__ j(equal, &place_float_param);

__ int3();

__ bind(&place_integer_param);
__ cmpq(start_int_section, current_int_param_slot);
// If there aren't any integer params remaining, just floats, then go to the
// next valuetype.
__ j(less_equal, &loop_through_valuetypes);

// Copy the param from the integer section to the actual parameter area.
__ pushq(MemOperand(current_int_param_slot, 0));
__ addq(current_int_param_slot, Immediate(kSystemPointerSize));
__ jmp(&loop_through_valuetypes);

__ bind(&place_float_param);
__ cmpq(start_float_section, current_float_param_slot);
// If there aren't any float params remaining, just integers, then go to the
// next valuetype.
__ j(less_equal, &loop_through_valuetypes);

// Copy the param from the float section to the actual parameter area.
__ pushq(MemOperand(current_float_param_slot, 0));
__ addq(current_float_param_slot, Immediate(kSystemPointerSize));
__ jmp(&loop_through_valuetypes);

__ bind(&params_done);
// Restore function_data after we are done with parameter placement.
function_data = rdi;
__ movq(function_data, MemOperand(rbp, kFunctionDataOffset));

__ bind(&prepare_for_wasm_call);
// -------------------------------------------
// Prepare for the Wasm call.
// -------------------------------------------
// Set thread_in_wasm_flag.
Register thread_in_wasm_flag_addr = r12;
__ movq(
    thread_in_wasm_flag_addr,
    MemOperand(kRootRegister, Isolate::thread_in_wasm_flag_address_offset()));
__ movl(MemOperand(thread_in_wasm_flag_addr, 0), Immediate(1));

Register jump_table_start = thread_in_wasm_flag_addr;
__ movq(jump_table_start,
        MemOperand(wasm_instance,
                   wasm::ObjectAccess::ToTagged(
                       WasmInstanceObject::kJumpTableStartOffset)));
thread_in_wasm_flag_addr = no_reg;

Register jump_table_offset = function_data;
__ LoadAnyTaggedField(
    jump_table_offset,
    MemOperand(
        function_data,
        WasmExportedFunctionData::kJumpTableOffsetOffset - kHeapObjectTag));

// Change from smi to integer.
__ SmiUntag(jump_table_offset);

Register function_entry = jump_table_offset;
__ addq(function_entry, jump_table_start);
jump_table_offset = no_reg;
jump_table_start = no_reg;

// We set the indicating value for the GC to the proper one for Wasm call.
constexpr int kWasmCallGCScanSlotCount = 0;
__ movq(MemOperand(rbp, kGCScanSlotCountOffset),
        Immediate(kWasmCallGCScanSlotCount));
```

```

// -------------------------------------------
// Call the Wasm function.
// -------------------------------------------
__ call(function_entry);
function_entry = no_reg;

// -------------------------------------------
// Resetting after the Wasm call.
// -------------------------------------------
// Restore rsp to free the reserved stack slots for the sections.
__ leaq(rsp, MemOperand(rbp, kLastSpillOffset));

// Unset thread_in_wasm_flag.
thread_in_wasm_flag_addr = r8;
__ movq(
    thread_in_wasm_flag_addr,
    MemOperand(kRootRegister, Isolate::thread_in_wasm_flag_address_offset()));
__ movl(MemOperand(thread_in_wasm_flag_addr, 0), Immediate(0));
thread_in_wasm_flag_addr = no_reg;

// -------------------------------------------
// Return handling.
// -------------------------------------------
return_count = r8;
__ movq(return_count, MemOperand(rbp, kReturnCountOffset));
Register return_reg = rax;

// If we have 1 return value, then jump to conversion.
__ cmpl(return_count, Immediate(1));
Label convert_return;
__ j(equal, &convert_return);

// Otherwise load undefined.
__ LoadRoot(return_reg, RootIndex::kUndefinedValue);

Label return_done;
__ bind(&return_done);
__ movq(param_count, MemOperand(rbp, kParamCountOffset));

// -------------------------------------------
// Deconstrunct the stack frame.
// -------------------------------------------
__ LeaveFrame(StackFrame::JS_TO_WASM);

// We have to remove the caller frame slots:
//  - JS arguments
//  - the receiver
// and transfer the control to the return address (the return address is
// expected to be on the top of the stack).
// We cannot use just the ret instruction for this, because we cannot pass the
// number of slots to remove in a Register as an argument.
Register return_addr = rbx;
__ popq(return_addr);
Register caller_frame_slots_count = param_count;
__ addq(caller_frame_slots_count, Immediate(1));
__ shlq(caller_frame_slots_count, Immediate(kSystemPointerSizeLog2));
__ addq(rsp, caller_frame_slots_count);
__ pushq(return_addr);
__ ret(0);

// ---------------------------------------------------------------------------
//                           Deferred code.
// ---------------------------------------------------------------------------

// -------------------------------------------
// Param conversion builtins.
// -------------------------------------------
__ bind(&convert_param);
// The order of pushes is important. We want the heap objects, that should be
```

```
546   // scanned by GC, to be on the top of the stack.
547   // We have to set the indicating value for the GC to the number of values on
548   // the top of the stack that have to be scanned before calling the builtin
549   // function.
550   // The builtin expects the parameter to be in register param = rax.
551
552   constexpr int kBuiltinCallGCScanSlotCount = 2;
553   PrepareForBuiltinCall(masm, MemOperand(rbp, kGCScanSlotCountOffset),
554                         kBuiltinCallGCScanSlotCount, current_param, param_limit,
555                         current_int_param_slot, current_float_param_slot,
556                         valuetypes_array_ptr, wasm_instance, function_data);
557
558   Label param_kWasmI32_not_smi;
559   Label param_kWasmI64;
560   Label param_kWasmF32;
561   Label param_kWasmF64;
562
563   __ cmpq(valuetype, Immediate(wasm::kWasmI32.raw_bit_field()));
564   __ j(equal, &param_kWasmI32_not_smi);
565
566   __ cmpq(valuetype, Immediate(wasm::kWasmI64.raw_bit_field()));
567   __ j(equal, &param_kWasmI64);
568
569   __ cmpq(valuetype, Immediate(wasm::kWasmF32.raw_bit_field()));
570   __ j(equal, &param_kWasmF32);
571
572   __ cmpq(valuetype, Immediate(wasm::kWasmF64.raw_bit_field()));
573   __ j(equal, &param_kWasmF64);
574
575   __ int3();
576
577   __ bind(&param_kWasmI32_not_smi);
578   __ Call(BUILTIN_CODE(masm->isolate(), WasmTaggedNonSmiToInt32),
579           RelocInfo::CODE_TARGET);
580   // Param is the result of the builtin.
581   __ AssertZeroExtended(param);
582   RestoreAfterBuiltinCall(masm, function_data, wasm_instance,
583                           valuetypes_array_ptr, current_float_param_slot,
584                           current_int_param_slot, param_limit, current_param);
585   __ movq(MemOperand(current_int_param_slot, 0), param);
586   __ subq(current_int_param_slot, Immediate(kSystemPointerSize));
587   __ jmp(&param_conversion_done);
588
589   __ bind(&param_kWasmI64);
590   __ Call(BUILTIN_CODE(masm->isolate(), BigIntToI64), RelocInfo::CODE_TARGET);
591   RestoreAfterBuiltinCall(masm, function_data, wasm_instance,
592                           valuetypes_array_ptr, current_float_param_slot,
593                           current_int_param_slot, param_limit, current_param);
594   __ movq(MemOperand(current_int_param_slot, 0), param);
595   __ subq(current_int_param_slot, Immediate(kSystemPointerSize));
596   __ jmp(&param_conversion_done);
597
598   __ bind(&param_kWasmF32);
599   __ Call(BUILTIN_CODE(masm->isolate(), WasmTaggedToFloat64),
600           RelocInfo::CODE_TARGET);
601   RestoreAfterBuiltinCall(masm, function_data, wasm_instance,
602                           valuetypes_array_ptr, current_float_param_slot,
603                           current_int_param_slot, param_limit, current_param);
604   // Clear higher bits.
605   __ Xorpd(xmm1, xmm1);
606   // Truncate float64 to float32.
607   __ Cvtsd2ss(xmm1, xmm0);
608   __ Movsd(MemOperand(current_float_param_slot, 0), xmm1);
609   __ subq(current_float_param_slot, Immediate(kSystemPointerSize));
610   __ jmp(&param_conversion_done);
611
612   __ bind(&param_kWasmF64);
613   __ Call(BUILTIN_CODE(masm->isolate(), WasmTaggedToFloat64),
614           RelocInfo::CODE_TARGET);
615   RestoreAfterBuiltinCall(masm, function_data, wasm_instance,
```

```
                          valuetypes_array_ptr, current_float_param_slot,
                          current_int_param_slot, param_limit, current_param);
  __ Movsd(MemOperand(current_float_param_slot, 0), xmm0);
  __ subq(current_float_param_slot, Immediate(kSystemPointerSize));
  __ jmp(&param_conversion_done);

  // ------------------------------------------
  // Return conversions.
  // ------------------------------------------
  __ bind(&convert_return);
  // We have to make sure that the kGCScanSlotCount is set correctly when we
  // call the builtins for conversion. For these builtins it's the same as for
  // the Wasm call, that is, kGCScanSlotCount = 0, so we don't have to reset it.
  // We don't need the JS context for these builtin calls.

  __ movq(valuetypes_array_ptr, MemOperand(rbp, kValueTypesArrayStartOffset));
  // The first valuetype of the array is the return's valuetype.
  __ movl(valuetype,
          Operand(valuetypes_array_ptr, wasm::ValueType::bit_field_offset()));

  Label return_kWasmI32;
  Label return_kWasmI64;
  Label return_kWasmF32;
  Label return_kWasmF64;

  __ cmpq(valuetype, Immediate(wasm::kWasmI32.raw_bit_field()));
  __ j(equal, &return_kWasmI32);

  __ cmpq(valuetype, Immediate(wasm::kWasmI64.raw_bit_field()));
  __ j(equal, &return_kWasmI64);

  __ cmpq(valuetype, Immediate(wasm::kWasmF32.raw_bit_field()));
  __ j(equal, &return_kWasmF32);

  __ cmpq(valuetype, Immediate(wasm::kWasmF64.raw_bit_field()));
  __ j(equal, &return_kWasmF64);

  __ int3();

  __ bind(&return_kWasmI32);
  Label to_heapnumber;
  // If pointer compression is disabled, we can convert the return to a smi.
  if (SmiValuesAre32Bits()) {
    __ SmiTag(return_reg);
  } else {
    Register temp = rbx;
    __ movq(temp, return_reg);
    // Double the return value to test if it can be a Smi.
    __ addl(temp, return_reg);
    temp = no_reg;
    // If there was overflow, convert the return value to a HeapNumber.
    __ j(overflow, &to_heapnumber);
    // If there was no overflow, we can convert to Smi.
    __ SmiTag(return_reg);
  }
  __ jmp(&return_done);

  // Handle the conversion of the I32 return value to HeapNumber when it cannot
  // be a smi.
  __ bind(&to_heapnumber);
  __ Call(BUILTIN_CODE(masm->isolate(), WasmInt32ToHeapNumber),
          RelocInfo::CODE_TARGET);
  __ jmp(&return_done);

  __ bind(&return_kWasmI64);
  __ Call(BUILTIN_CODE(masm->isolate(), I64ToBigInt), RelocInfo::CODE_TARGET);
  __ jmp(&return_done);

  __ bind(&return_kWasmF32);
  // The builtin expects the value to be in xmm0.
```

```
  __ Movss(xmm0, xmm1);
  __ Call(BUILTIN_CODE(masm->isolate(), WasmFloat32ToNumber),
          RelocInfo::CODE_TARGET);
  __ jmp(&return_done);

  __ bind(&return_kWasmF64);
  // The builtin expects the value to be in xmm0.
  __ Movsd(xmm0, xmm1);
  __ Call(BUILTIN_CODE(masm->isolate(), WasmFloat64ToNumber),
          RelocInfo::CODE_TARGET);
  __ jmp(&return_done);
}
```

Source code for the GC support

```
// Copyright 2012 the V8 project authors. All rights reserved.
// Use of this source code is governed by a BSD-style license that can be
// found in the LICENSE file.

void JsToWasmFrame::Iterate(RootVisitor* v) const {
  Code code = GetContainingCode(isolate(), pc());
  //  GenericJSToWasmWrapper stack layout
  //  ------+-----------------+----------------------
  //        |  return addr    |
  //    fp  |- - - - - - - - -|  -------------------|
  //        |       fp        |                     |
  //   fp-p |- - - - - - - - -|                     |
  //        |  frame marker   |                     | no GC scan
  //  fp-2p |- - - - - - - - -|                     |
  //        |   scan_count    |                     |
  //  fp-3p |- - - - - - - - -|  -------------------|
  //        |      ....       | <- spill_slot_limit |
  //        |   spill slots   |                     | GC scan scan_count slots
  //        |      ....       | <- spill_slot_base--|
  //        |- - - - - - - - -|                     |
  if (code.is_null() || !code.is_builtin() ||
      code.builtin_index() != Builtins::kGenericJSToWasmWrapper) {
    // If it's not the  GenericJSToWasmWrapper, then it's the TurboFan compiled
    // specific wrapper. So we have to call IterateCompiledFrame.
    IterateCompiledFrame(v);
    return;
  }
  // The [fp - 2*kSystemPointerSize] on the stack is a value indicating how
  // many values should be scanned from the top.
  intptr_t scan_count =
      *reinterpret_cast<intptr_t*>(fp() - 2 * kSystemPointerSize);

  FullObjectSlot spill_slot_base(&Memory<Address>(sp()));
  FullObjectSlot spill_slot_limit(
      &Memory<Address>(sp() + scan_count * kSystemPointerSize));
  v->VisitRootPointers(Root::kTop, nullptr, spill_slot_base, spill_slot_limit);
}
```

Source code of the tests

```
// Copyright 2020 the V8 project authors. All rights reserved.
// Use of this source code is governed by a BSD-style license that can be
// found in the LICENSE file.

// Flags: --wasm-generic-wrapper --expose-gc --allow-natives-syntax

load("test/mjsunit/wasm/wasm-module-builder.js");

(function testGenericWrapper0Param() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_v_v);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprCallFunction, func_index
    ])
    .exportFunc();

  let x = 12;
  function import_func() {
    gc();
    x = 20;
  }

  let instance = builder.instantiate({ mod: { func: import_func } });
  assertEquals(undefined, instance.exports.main());
  assertEquals(x, 20);
})();

(function testGenericWrapper0ParamTraps() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_v_v);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprUnreachable
    ])
    .exportFunc();

  let instance = builder.instantiate();
  assertTraps(kTrapUnreachable, instance.exports.main);
})();

(function testGenericWrapper1ParamTrap() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_v_i);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprLocalGet, 0, kExprUnreachable
    ])
    .exportFunc();

  let instance = builder.instantiate();
```

```
  assertTraps(kTrapUnreachable, () => instance.exports.main(1));
})();

(function testGenericWrapper1ParamGeneral() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_v_i);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprLocalGet, 0, kExprCallFunction, func_index
    ])
    .exportFunc();

  let x = 12;
  function import_func(param) {
    gc();
    x += param;
  }

  let instance = builder.instantiate({ mod: { func: import_func } });
  assertEquals(undefined, instance.exports.main(5));
  assertEquals(17, x);
})();

(function testGenericWrapper1ParamNotSmi() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_v_i);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprLocalGet, 0, kExprCallFunction, func_index
    ])
    .exportFunc();

  let x = 12;
  function import_func(param) {
    gc();
    x += param;
  }

  let y = { valueOf: () => { print("Hello!"); gc(); return 24; } };
  let instance = builder.instantiate({ mod: { func: import_func } });
  assertEquals(undefined, instance.exports.main(y));
  assertEquals(36, x);
})();

(function testGenericWrapper4Param() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_v_iiii);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprLocalGet, 0,
      kExprLocalGet, 1,
      kExprLocalGet, 2,
      kExprLocalGet, 3,
      kExprCallFunction, func_index
    ])
    .exportFunc();

  let x = 12;
  function import_func(param1, param2, param3, param4) {
    gc();
    x += 2 * param1 + 3 * param2 + 4 * param3 + 5 * param4;
  }

  let param2 = { valueOf: () => { gc(); return 6; } };
```

```
  let param3 = { valueOf: () => { gc(); return 3; } };
  let instance = builder.instantiate({ mod: { func: import_func } });
  assertEquals(undefined, instance.exports.main(9, param2, param3, 0));
  assertEquals(60, x);
  // Now we test if the evaluation order of the parameters is correct.
  x = 12;
  param3 = {
    valueOf: () => {
      Object.defineProperty(param2, 'valueOf', {
        value: () => 30
      })
      return 3;
    }
  };
  assertEquals(undefined, instance.exports.main(9, param2, param3, 0));
  assertEquals(60, x);
})();

let kSig_v_iiiiiiii = makeSig([kWasmI32, kWasmI32, kWasmI32, kWasmI32,
  kWasmI32, kWasmI32, kWasmI32, kWasmI32], []);

(function testGenericWrapper8Param() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_v_iiiiiiii);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprLocalGet, 0,
      kExprLocalGet, 1,
      kExprLocalGet, 2,
      kExprLocalGet, 3,
      kExprLocalGet, 4,
      kExprLocalGet, 5,
      kExprLocalGet, 6,
      kExprLocalGet, 7,
      kExprCallFunction, func_index
    ])
    .exportFunc();

  let x = 12;
  function import_func(param1, param2, param3, param4, param5, param6,
    param7, param8) {
    gc();
    x += param1 + 2 * param2 + 3 * param3 + 4 * param4 + 5 * param5
      + 6 * param6 + 7 * param7 + 8 * param8;
  }

  let param1 = { valueOf: () => { gc(); return 5; } };
  let param4 = { valueOf: () => { gc(); return 8; } };
  let param6 = { valueOf: () => { gc(); return 10; } };
  let param8 = { valueOf: () => { gc(); return 12; } };
  let instance = builder.instantiate({ mod: { func: import_func } });
  assertEquals(undefined, instance.exports.main(param1, 6, 7, param4, 9, param6, 11, ↩
      param8));
  assertEquals(360, x);
})();

// Passing less parameters than expected.
(function testGenericWrapper4ParamWithLessParams() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_v_iiii);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprLocalGet, 0,
      kExprLocalGet, 1,
      kExprLocalGet, 2,
      kExprLocalGet, 3,
```

```
      kExprCallFunction, func_index
    ])
    .exportFunc();

  let x = 12;
  function import_func(param1, param2, param3, param4) {
    gc();
    x += param1 + param2 + param3 + param4;
  }

  let param2 = { valueOf: () => { gc(); return 3; } };
  let instance = builder.instantiate({ mod: { func: import_func } });
  assertEquals(undefined, instance.exports.main(5, param2));
  assertEquals(20, x);
})();

// Passing more parameters than expected.
(function testGenericWrapper4ParamWithMoreParams() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_v_iiii);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprLocalGet, 0,
      kExprLocalGet, 1,
      kExprLocalGet, 2,
      kExprLocalGet, 3,
      kExprCallFunction, func_index
    ])
    .exportFunc();

  let x = 12;
  function import_func(param1, param2, param3, param4) {
    gc();
    x += param1 + param2 + param3 + param4;
  }

  let param2 = { valueOf: () => { gc(); return 3; } };
  let param3 = { valueOf: () => { gc(); return 6; } };
  let instance = builder.instantiate({ mod: { func: import_func } });
  assertEquals(undefined, instance.exports.main(5, param2, param3, 7, 200, 300, 400));
  assertEquals(33, x);
})();

(function testGenericWrapper1I32ReturnSmi() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_i_i);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprLocalGet, 0, kExprCallFunction, func_index
    ])
    .exportFunc();

  let x = 12;
  function import_func(param) {
    gc();
    return x + param;
  }

  let instance = builder.instantiate({ mod: { func: import_func } });
  assertEquals(17, instance.exports.main(5));
})();

(function testGenericWrapper1I32ReturnHeapNumber() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_i_i);
```

```
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprLocalGet, 0, kExprCallFunction, func_index
    ])
    .exportFunc();

  let x = 2147483640;
  function import_func(param) {
    let result = x + param;
    %SimulateNewspaceFull();
    return result;
  }

  let instance = builder.instantiate({ mod: { func: import_func } });
  assertEquals(2147483645, instance.exports.main(5));
})();

let kSig_i_lili = makeSig([kWasmI64, kWasmI32, kWasmI64, kWasmI32], [kWasmI32]);

(function testGenericWrapper4IParam1I32Ret() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_i_lili);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprLocalGet, 0,
      kExprLocalGet, 1,
      kExprLocalGet, 2,
      kExprLocalGet, 3,
      kExprCallFunction, func_index
    ])
    .exportFunc();

  let x = 12n;
  function import_func(param1, param2, param3, param4) {
    x += 2n * param1 + BigInt(3 * param2) + 4n * param3 + BigInt(5 * param4);
    return Number(x);
  }

  let param2 = { valueOf: () => { gc(); return 6; } };
  let param3 = { valueOf: () => { gc(); return 3n; } };
  let instance = builder.instantiate({ mod: { func: import_func } });
  assertEquals(60, instance.exports.main(9n, param2, param3, 0));
})();

let kSig_v_iiili = makeSig([kWasmI32, kWasmI32, kWasmI32, kWasmI64,
  kWasmI32], []);

(function testGenericWrapper5IParam() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_v_iiili);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprLocalGet, 0,
      kExprLocalGet, 1,
      kExprLocalGet, 2,
      kExprLocalGet, 3,
      kExprLocalGet, 4,
      kExprCallFunction, func_index
    ])
    .exportFunc();

  let x = 12;
  function import_func(param1, param2, param3, param4, param5) {
    gc();
    x += 2 * param1 + 3 * param2 + 4 * param3 + 5 * Number(param4) + 6 * param5;
```

```
  }

  let param2 = { valueOf: () => { gc(); return 6; } };
  let param3 = { valueOf: () => { gc(); return 3; } };
  let instance = builder.instantiate({ mod: { func: import_func } });
  assertEquals(undefined, instance.exports.main(9, param2, param3, 0n, 2));
  assertEquals(72, x);
})();

let kSig_v_iiiiii = makeSig([kWasmI32, kWasmI32, kWasmI32, kWasmI64,
  kWasmI32, kWasmI32], []);

(function testGenericWrapper6IParam() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_v_iiiiii);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprLocalGet, 0,
      kExprLocalGet, 1,
      kExprLocalGet, 2,
      kExprLocalGet, 3,
      kExprLocalGet, 4,
      kExprLocalGet, 5,
      kExprCallFunction, func_index
    ])
    .exportFunc();

  let x = 12;
  function import_func(param1, param2, param3, param4, param5, param6) {
    gc();
    x += 2 * param1 + 3 * param2 + 4 * param3 + 5 * Number(param4) + 6 * param5 + 7 * ↩
        param6;
  }

  let param2 = { valueOf: () => { gc(); return 6; } };
  let param3 = { valueOf: () => { gc(); return 3; } };
  let instance = builder.instantiate({ mod: { func: import_func } });
  assertEquals(undefined, instance.exports.main(9, param2, param3, 0n, 2, 3));
  assertEquals(93, x);
})();

let kSig_v_liiiiiiiiil = makeSig([kWasmI64, kWasmI32, kWasmI32, kWasmI64,
  kWasmI64, kWasmI32, kWasmI32, kWasmI32, kWasmI64], [kWasmI32]);

(function testGenericWrapper9IParam132Ret() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_v_liiiiiiiil);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprLocalGet, 0,
      kExprLocalGet, 1,
      kExprLocalGet, 2,
      kExprLocalGet, 3,
      kExprLocalGet, 4,
      kExprLocalGet, 5,
      kExprLocalGet, 6,
      kExprLocalGet, 7,
      kExprLocalGet, 8,
      kExprCallFunction, func_index
    ])
    .exportFunc();

  let x = 12;
  function import_func(param1, param2, param3, param4, param5, param6,
    param7, param8, param9) {
```

```
    x += Number(param1) + 2 * param2 + 3 * param3 + Number(4n * param4) + Number(5n * param5)
      + 6 * param6 + 7 * param7 + 8 * param8 + Number(9n * param9);
    return x;
  }

  let param1 = { valueOf: () => { gc(); return 5n; } };
  let param4 = { valueOf: () => { gc(); return 8n; } };
  let param6 = { valueOf: () => { gc(); return 10; } };
  let param8 = { valueOf: () => { gc(); return 12; } };
  let instance = builder.instantiate({ mod: { func: import_func } });
  assertEquals(360, instance.exports.main(param1, 6, 7, param4, 9n, param6, 11, param8, 0n));
})();

// The function expects BigInt, but gets Number.
(function testGenericWrapperTypeError() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_v_l);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprLocalGet, 0, kExprCallFunction, func_index
    ])
    .exportFunc();

  let x = 12n;
  function import_func(param1) {
    x += param1;
  }

  let instance = builder.instantiate({ mod: { func: import_func } });
  assertThrows(() => { instance.exports.main(17) }, TypeError);
})();

(function testGenericWrapper1I64Return() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_l_v);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprCallFunction, func_index
    ])
    .exportFunc();

  function import_func() {
    gc();
    return 10000000000n;
  }

  let instance = builder.instantiate({ mod: { func: import_func } });
  assertEquals(10000000000n, instance.exports.main());
})();

(function testGenericWrapper1F32Return() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_f_v);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprCallFunction, func_index
    ])
    .exportFunc();

  function import_func() {
    gc();
    return 0.5;
```

```
  }

  let instance = builder.instantiate({ mod: { func: import_func } });
  assertEquals(0.5, instance.exports.main());
})();

(function testGenericWrapper1F64Return() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_d_v);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprCallFunction, func_index
    ])
    .exportFunc();

  function import_func() {
    gc();
    return 0.25;
  }

  let instance = builder.instantiate({ mod: { func: import_func } });
  assertEquals(0.25, instance.exports.main());
})();

(function testGenericWrapper1Float32() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_v_f);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprLocalGet, 0, kExprCallFunction, func_index
    ])
    .exportFunc();

  let x = 12.5;
  function import_func(param) {
    gc();
    x += param;
  }

  let instance = builder.instantiate({ mod: { func: import_func } });
  assertEquals(undefined, instance.exports.main(12.5));
  assertEquals(25, x);
})();

(function testGenericWrapper1Float64() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_v_d);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprLocalGet, 0, kExprCallFunction, func_index
    ])
    .exportFunc();

  let x = 12.5;
  function import_func(param) {
    gc();
    x += param;
  }

  let instance = builder.instantiate({ mod: { func: import_func } });
  assertEquals(undefined, instance.exports.main(12.5));
  assertEquals(25, x);
})();
```

```
let kSig_v_ffddddff = makeSig([kWasmF32, kWasmF32, kWasmF64, kWasmF64,
  kWasmF64, kWasmF64, kWasmF32, kWasmF32], []);

(function testGenericWrapper8Floats() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_v_ffddddff);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprLocalGet, 0,
      kExprLocalGet, 1,
      kExprLocalGet, 2,
      kExprLocalGet, 3,
      kExprLocalGet, 4,
      kExprLocalGet, 5,
      kExprLocalGet, 6,
      kExprLocalGet, 7,
      kExprCallFunction, func_index
    ])
    .exportFunc();

  let x = 12;
  function import_func(param1, param2, param3, param4, param5, param6,
    param7, param8) {
    gc();
    x += param1 + 2 * param2 + 3 * param3 + 4 * param4 + 5 * param5
      + 6 * param6 + 7 * param7 + 8 * param8;
  }

  let param1 = { valueOf: () => { gc(); return 1.5; } };
  let param4 = { valueOf: () => { gc(); return 4.5; } };
  let param6 = { valueOf: () => { gc(); return 6.5; } };
  let param8 = { valueOf: () => { gc(); return 8.5; } };
  let instance = builder.instantiate({ mod: { func: import_func } });
  assertEquals(undefined, instance.exports.main(param1, 2.5, 3.5, param4,
    5.5, param6, 7.5, param8));
  assertEquals(234, x);
})();

let kSig_v_iiliffddlfdff = makeSig([kWasmI32, kWasmI32, kWasmI64,
  kWasmI32, kWasmF32, kWasmF32, kWasmF64, kWasmF64, kWasmI64, kWasmF32,
  kWasmF64, kWasmF32, kWasmF32], []);
// Floats don't fit into param registers.
(function testGenericWrapper13ParamMix() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_v_iiliffddlfdff);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprLocalGet, 0,
      kExprLocalGet, 1,
      kExprLocalGet, 2,
      kExprLocalGet, 3,
      kExprLocalGet, 4,
      kExprLocalGet, 5,
      kExprLocalGet, 6,
      kExprLocalGet, 7,
      kExprLocalGet, 8,
      kExprLocalGet, 9,
      kExprLocalGet, 10,
      kExprLocalGet, 11,
      kExprLocalGet, 12,
      kExprCallFunction, func_index
    ])
    .exportFunc();

  let x = 12;
  let y = 1.0;
```

```
611   function import_func(parami1, parami2, paraml1, parami3, paramf1, paramf2,
612     paramd1, paramd2, paraml2, paramf3, paramd3, paramf4, paramf5) {
613     gc();
614     x += parami1 + 2 * parami2 + 3 * Number(paraml1) + 4 * parami3
615       + 5 * Number(paraml2);
616     y += paramf1 + 2 * paramf2 + 3 * paramd1 + 4 * paramd2 + 5 * paramf3
617       + 6 * paramd3 + 7 * paramf4 + 8 * paramf5;
618   }
619
620   let instance = builder.instantiate({ mod: { func: import_func } });
621   assertEquals(undefined, instance.exports.main(5, 6, 7n, 8, 1.5, 2.5, 3.5,
622     4.5, 11n, 5.5, 6.5, 7.5, 8.5));
623   assertEquals(137, x);
624   assertEquals(223, y);
625 })();
626
627 let kSig_v_iiiiiiffddli = makeSig([kWasmI32, kWasmI32, kWasmI64, kWasmI32,
628   kWasmI32, kWasmI32, kWasmF32, kWasmF32, kWasmF64, kWasmF64, kWasmI64,
629   kWasmI32], []);
630 // Integers don't fit into param registers.
631 (function testGenericWrapper12ParamMix() {
632   print(arguments.callee.name);
633   let builder = new WasmModuleBuilder();
634   let sig_index = builder.addType(kSig_v_iiiiiiffddli);
635   let func_index = builder.addImport("mod", "func", sig_index);
636   builder.addFunction("main", sig_index)
637     .addBody([
638       kExprLocalGet, 0,
639       kExprLocalGet, 1,
640       kExprLocalGet, 2,
641       kExprLocalGet, 3,
642       kExprLocalGet, 4,
643       kExprLocalGet, 5,
644       kExprLocalGet, 6,
645       kExprLocalGet, 7,
646       kExprLocalGet, 8,
647       kExprLocalGet, 9,
648       kExprLocalGet, 10,
649       kExprLocalGet, 11,
650       kExprCallFunction, func_index
651     ])
652     .exportFunc();
653
654   let x = 12;
655   let y = 1.0;
656   function import_func(param1, param2, param3, param4, param5, param6,
657     paramf1, paramf2, paramd1, paramd2, param7, param8) {
658     gc();
659     x += param1 + 2 * param2 + 3 * Number(param3) + 4 * param4 + 5 * param5
660       + 6 * param6 + 7 * Number(param7) + 8 * param8;
661     y += paramf1 + paramf2 + paramd1 + paramd2;
662   }
663
664   let param1 = { valueOf: () => { gc(); return 5; } };
665   let param4 = { valueOf: () => { gc(); return 8; } };
666   let param6 = { valueOf: () => { gc(); return 10; } };
667   let param8 = { valueOf: () => { gc(); return 12; } };
668   let instance = builder.instantiate({ mod: { func: import_func } });
669   assertEquals(undefined, instance.exports.main(param1, 6, 7n, param4, 9, param6,
670     1.5, 2.5, 3.6, 4.4, 11n, param8));
671   assertEquals(360, x);
672   assertEquals(13, y);
673 })();
674
675 let kSig_f_iiiiiiffddlifffdi = makeSig([kWasmI32, kWasmI32, kWasmI64, kWasmI32,
676   kWasmI32, kWasmI32, kWasmF32, kWasmF32, kWasmF64, kWasmF64, kWasmI64,
677   kWasmI32, kWasmF32, kWasmF32, kWasmF32, kWasmF64, kWasmI32], [kWasmF32]);
678 // Integers and floats don't fit into param registers.
679 (function testGenericWrapper17ParamMix() {
680   print(arguments.callee.name);
```

```
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_f_iiiiiiffddlifffdi);
  let func_index = builder.addImport("mod", "func", sig_index);
  builder.addFunction("main", sig_index)
    .addBody([
      kExprLocalGet, 0,
      kExprLocalGet, 1,
      kExprLocalGet, 2,
      kExprLocalGet, 3,
      kExprLocalGet, 4,
      kExprLocalGet, 5,
      kExprLocalGet, 6,
      kExprLocalGet, 7,
      kExprLocalGet, 8,
      kExprLocalGet, 9,
      kExprLocalGet, 10,
      kExprLocalGet, 11,
      kExprLocalGet, 12,
      kExprLocalGet, 13,
      kExprLocalGet, 14,
      kExprLocalGet, 15,
      kExprLocalGet, 16,
      kExprCallFunction, func_index
    ])
    .exportFunc();

  let x = 12;
  function import_func(param1, param2, param3, param4, param5, param6,
    paramf1, paramf2, paramd1, paramd2, param7, param8, paramf3, paramf4,
    paramf5, paramd3, param9) {
    gc();
    x += param1 + 2 * param2 + 3 * Number(param3) + 4 * param4 + 5 * param5
      + 6 * param6 + 7 * Number(param7) + 8 * param8 + 9 * param9;
    let y = 1.0;
    y += paramf1 + 2 * paramf2 + 3 * paramd1 + 4 * paramd2 + 5 * paramf3
      + 6 * paramf4 + 7 * paramf5 + 8 * paramd3;
    return y;
  }

  let param1 = { valueOf: () => { gc(); return 5; } };
  let param4 = { valueOf: () => { gc(); return 8; } };
  let param6 = { valueOf: () => { gc(); return 10; } };
  let param8 = { valueOf: () => { gc(); return 12; } };
  let paramd1 = { valueOf: () => { gc(); return 3.5; } };
  let paramf3 = { valueOf: () => { gc(); return 5.5; } };
  let param9 = { valueOf: () => { gc(); return 0; } };
  let instance = builder.instantiate({ mod: { func: import_func } });
  assertEquals(223, instance.exports.main(param1, 6, 7n, param4, 9, param6,
    1.5, 2.5, paramd1, 4.5, 11n, param8, paramf3, 6.5, 7.5, 8.5, param9));
  assertEquals(360, x);
})();

(function Regression1130385() {
  print(arguments.callee.name);
  let builder = new WasmModuleBuilder();
  let sig_index = builder.addType(kSig_i_v);
  builder.addFunction("f0", sig_index)
    .addBody([kExprI32Const, 12])
    .exportFunc();

  builder.addFunction("f1", sig_index)
    .addBody([kExprI32Const, 15])
    .exportFunc();

  let instance = builder.instantiate();
  assertEquals(15, instance.exports.f1());
})();
```