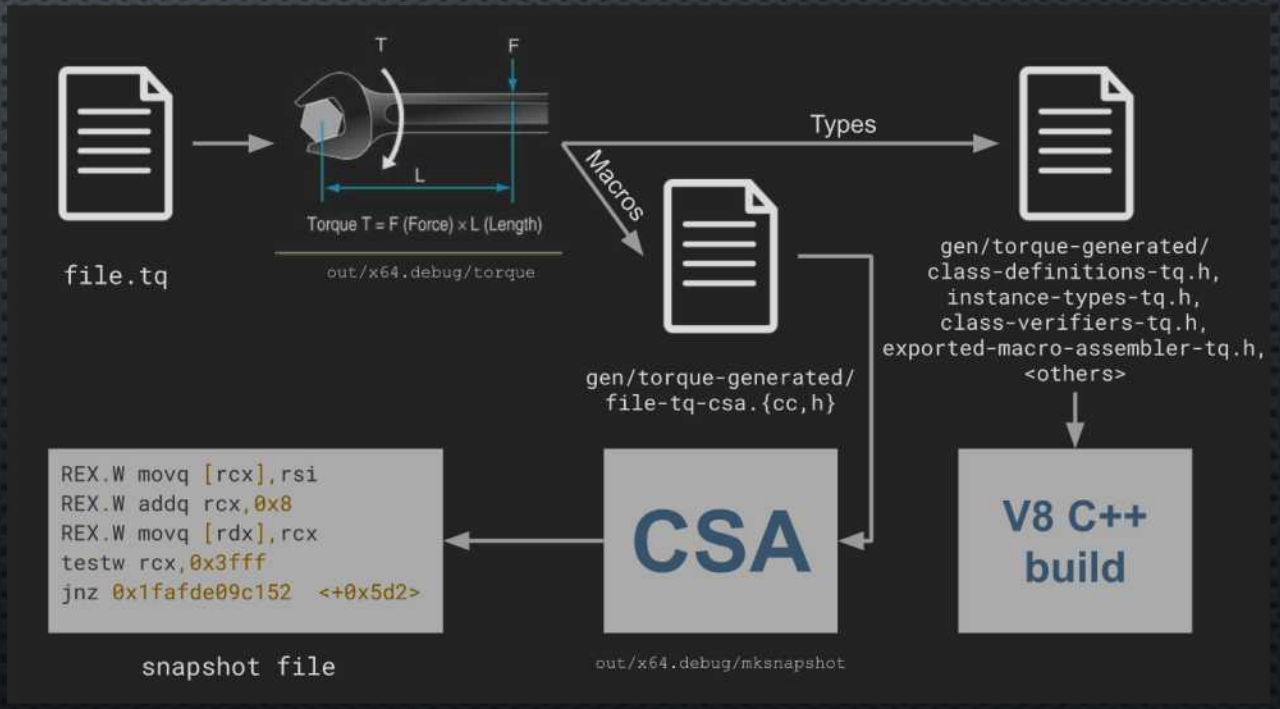
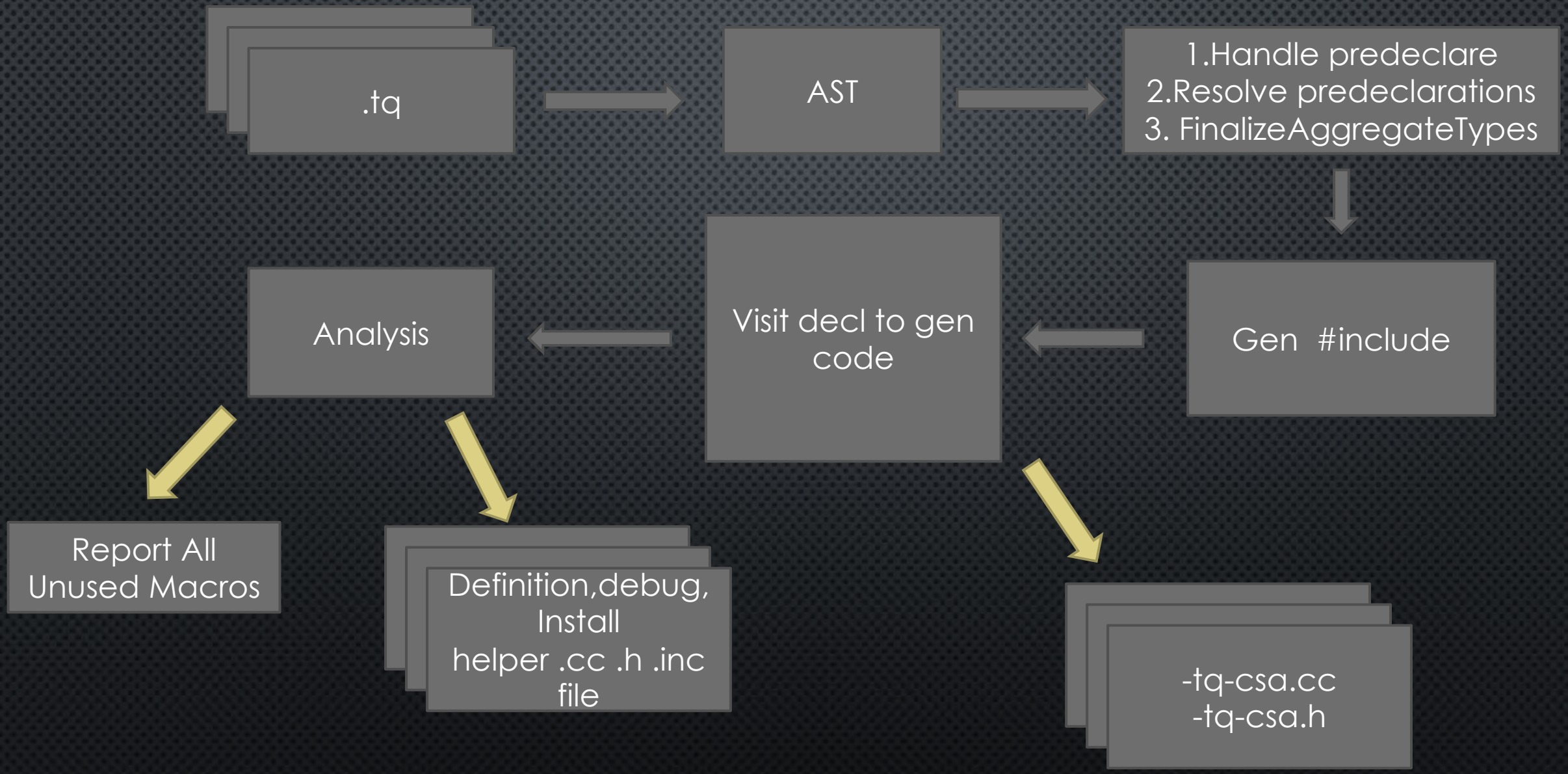
Dive Into Torque

v8.dev中对于csa部分文档的缺失，导致开发者无法系统的学习csa。为了更好的完成任务，决定从torque编译器入手，分析tq ->csa这个过程，来学习csa的各种功能。

1.Torque 编译流程
2.中端对于transient和explicit的处理
3.从bb到csa的生成过程




.tq
AST
1.Handle predeclare
2.Resolve predeclarations
3. FinalizeAggregateTypes
Gen #include
Visit decl to gen code
Analysis
Report All Unused Macros
Definition,debug, Install helper .cc .h .inc file
-tq-csa.cc
-tq-csa.h

Csa文件的依赖生成

```
void ImplementationVisitor::BeginCSAFiles() {
  for (SourceId file : SourceFileMap::AllSources()) {
    std::ostream& source = GlobalContext::GeneratedPerFile(file).csa_ccfile;
    std::ostream& header = GlobalContext::GeneratedPerFile(file).csa_headerfile;

    for (const std::string& include_path : GlobalContext::CppIncludes()) {
      source << "#include " << StringLiteralQuote(include_path) << "\n";
    }

    for (SourceId file : SourceFileMap::AllSources()) {
      source << "#include \"torque-generated/" +
                    SourceFileMap::PathFromV8RootWithoutExtension(file) +
                    "-tq-csa.h\"\n";
    }
    source << "\n";

    source << "namespace v8 {\n"
           << "namespace internal {\n"
           << "\n";

    std::string headerDefine =
        "V8_GEN_TORQUE_GENERATED_" +
        UnderlinifyPath(SourceFileMap::PathFromV8Root(file)) + "_H_";
    header << "#ifndef " << headerDefine << "\n";
    header << "#define " << headerDefine << "\n\n";
    header << "#include \"src/builtins/builtins-promise.h\"\n";
    header << "#include \"src/compiler/code-assembler.h\"\n";
    header << "#include \"src/codegen/code-stub-assembler.h\"\n";
    header << "#include \"src/utils/utils.h\"\n";
    header << "#include \"torque-generated/field-offsets-tq.h\"\n";
    header << "#include \"torque-generated/csa-types-tq.h\"\n";
    header << "\n";

    header << "namespace v8 {\n"
           << "namespace internal {\n"
           << "\n";
  }
}
```

```
#include "src/builtins/builtins-string-gen.h"
#include "src/builtins/builtins-typed-array-gen.h"
#include "src/builtins/builtins-utils-gen.h"
#include "src/builtins/builtins.h"
#include "src/codegen/code-factory.h"
#include "src/heap/factory-inl.h"
#include "src/objects/arguments.h"
#include "src/objects/bigint.h"
#include "src/objects/elements-kind.h"
#include "src/objects/free-space.h"
#include "src/objects/js-aggregate-error.h"
#include "src/objects/js-break-iterator.h"
#include "src/objects/js-collator.h"
#include "src/objects/js-date-time-format.h"
#include "src/objects/js-display-names.h"
#include "src/objects/js-generator.h"
#include "src/objects/js-list-format.h"
#include "src/objects/js-locale.h"
#include "src/objects/js-number-format.h"
#include "src/objects/js-objects.h"
#include "src/objects/js-plural-rules.h"
#include "src/objects/js-promise.h"
#include "src/objects/js-regexp-string-iterator.h"
#include "src/objects/js-relative-time-format.h"
#include "src/objects/js-segment-iterator.h"
#include "src/objects/js-segmenter.h"
#include "src/objects/js-weak-refs.h"
#include "src/objects/objects.h"
#include "src/objects/ordered-hash-table.h"
#include "src/objects/property-array.h"
#include "src/objects/property-descriptor-object.h"
#include "src/objects/source-text-module.h"
#include "src/objects/stack-frame-info.h"
#include "src/objects/synthetic-module.h"
#include "src/objects/template-objects.h"
#include "src/torque/runtime-support.h"
#include "torque-generated/src/builtins/array-copywithin-tq-csa.h"
#include "torque-generated/src/builtins/array-every-tq-csa.h"
```

Visit decl to gen code

```
}
switch (declarable->kind()) {
  case Declarable::kExternMacro:
    return Visit(ExternMacro::cast(declarable));
  case Declarable::kTorqueMacro:
    return Visit(TorqueMacro::cast(declarable));
  case Declarable::kMethod:
    return Visit(Method::cast(declarable));
  case Declarable::kBuiltin:
    return Visit(Builtin::cast(declarable));
  case Declarable::kTypeAlias:
    return Visit(TypeAlias::cast(declarable));
  case Declarable::kNamespaceConstant:
    return Visit(NamespaceConstant::cast(declarable));
  case Declarable::kRuntimeFunction:
  case Declarable::kIntrinsic:
  case Declarable::kExternConstant:
  case Declarable::kNamespace:
  case Declarable::kGenericCallable:
  case Declarable::kGenericType:
    return;
}
```

1 call | 1 ref

```
void ImplementationVisitor::Visit(TorqueMacro* macro) {
  VisitMacroCommon(macro);
}
```

1 call | 1 ref

```
void ImplementationVisitor::Visit(Method* method) {
  DCHECK(!method->IsExternal());
  VisitMacroCommon(method);
}
```

1 call | 15 refs

```
void ImplementationVisitor::Visit(Builtin* builtin) {
  if (builtin->IsExternal()) return;
  CurrentScope::Scope current_scope(builtin);
  CurrentCallable::Scope current_callable(builtin);
  CurrentReturnValue::Scope current_return_value;

  const std::string& name = builtin->ExternalName();
  const Signature& signature = builtin->signature();
  source_out() << "TF_BUILTIN(" << name << ", CodeStubAssembler) {\n"
               << "  compiler::CodeAssemblerState* state_ = state();"
               << "  compiler::CodeAssembler ca_(state());\n";
```

1 call | 4 refs

```
void ImplementationVisitor::Visit(NamespaceConstant* decl) {
  Signature signature{{}, base::nullopt, {{}, false}, 0, decl->type(),
                      {}, false};

  BindingsManagersScope bindings_managers_scope;
```

```
for (size_t i = 0; i < signature.implicit_count; ++i) {
  const std::string& param_name = signature.parameter_names[i]->value;
  SourcePosition param_pos = signature.parameter_names[i]->pos;
  std::string generated_name = AddParameter(
      i, builtin, &parameters, &parameter_types, &parameter_bindings, true);
  const Type* actual_type = signature.parameter_types.types[i];
  std::vector<const Type*> expected_types;
  if (param_name == "context") {
    source_out() << "  TNode<NativeContext> " << generated_name
                 << " = UncheckedCast<NativeContext>(Parameter("
                 << "Descriptor::kContext));\n";
    source_out() << "  USE(" << generated_name << ");\n";
    expected_types = {TypeOracle::GetNativeContextType(),
                      TypeOracle::GetContextType()};
  } else if (param_name == "receiver") {
    source_out()
        << "  TNode<Object> " << generated_name << " = "
        << (builtin->IsVarArgsJavaScript()
                ? "arguments.GetReceiver()"
                : "UncheckedCast<Object>(Parameter(Descriptor::kReceiver))")
        << ";\n";
    source_out() << "USE(" << generated_name << ");\n";
    expected_types = {TypeOracle::GetJSAnyType()};
  } else if (param_name == "newTarget") {
    source_out() << "  TNode<Object> " << generated_name
                 << " = UncheckedCast<Object>(Parameter("
                 << "Descriptor::kJSNewTarget));\n";
    source_out() << "USE(" << generated_name << ");\n";
    expected_types = {TypeOracle::GetJSAnyType()};
  } else if (param_name == "target") {
    source_out() << "  TNode<JSFunction> " << generated_name
                 << " = UncheckedCast<JSFunction>(Parameter("
                 << "Descriptor::kJSTarget));\n";
    source_out() << "USE(" << generated_name << ");\n";
    expected_types = {TypeOracle::GetJSFunctionType()};
  } else {
    Error(
        "Unexpected implicit parameter \"", param_name,
        "\" for JavaScript calling convention, "
        "expected \"context\", \"receiver\", \"target\", or \"newTarget\"")
        .Position(param_pos);
    expected_types = {actual_type};
  }
```

js builtin 中的 Explicit parameters处理

Explicit parameters

Declarations of Torque-defined Callables, e.g. Torque `macros` and `builtins`, have explicit parameter lists. They are a list of identifier and type pairs using a syntax reminiscent of typed TypeScript function parameter lists, with the exception that Torque doesn't support optional parameters or default parameters. Moreover, Torque-implement `builtins` can optonally support rest parameters if the builtin uses V8's internal JavaScript calling convention (e.g. is marked with the `javascript` keyword).

ExplicitParameters :
  ((*IdentifierName* : *TypeIdentifierName*)$_{list*}$)
  ((*IdentifierName* : *TypeIdentifierName*)$_{list+}$ (, ... *IdentifierName*)$_{opt}$)

As an example:

```
javascript builtin ArraySlice(
    (implicit context: Context)(receiver: Object, ...arguments): Object {
  // …
}
```

Transient 类型的处理

```
const fastArray : FastJSArray = Cast<FastJSArray>(array) otherwise Bailout;
Call(f, Undefined);
return fastArray; // Type error: fastArray is invalid here.
```

```
30 calls | 2 refs
void Emit(Instruction instruction) {
  instruction.TypeInstruction(&current_stack_, &cfg_);
  current_block_->Add(std::move(instruction));
}
```

```
6 calls
void InstructionBase::InvalidateTransientTypes(
    2 refs
    Stack<const Type*>* stack) const {
  auto current = stack->begin();
  while (current != stack->end()) {
    if ((*current)->IsTransient()) {
      std::stringstream stream;
      stream << "type " << **current
             << " is made invalid by transitioning callable invocation at "
             << PositionAsString(pos);
      *current = TypeOracle::GetTopType(stream.str(), *current);
    }
    ++current;
  }
}
```

重载EmitInstruction，对每个指令生成csa

```
3 calls
17  base::Optional<Stack<std::string>> CSAGenerator::EmitGraph(
67  }
68
2 calls | 6 refs
69  Stack<std::string> CSAGenerator::EmitBlock(const Block* block) {…
89  }
90
0 calls | 4 refs | 1 ref
91  void CSAGenerator::EmitSourcePosition(SourcePosition pos, bool always_emit) {…
100 }
101
0 calls | 1 ref
102 bool CSAGenerator::IsEmptyInstruction(const Instruction& instruction) {…
114 }
115
1 call | 2 refs
116 void CSAGenerator::EmitInstruction(const Instruction& instruction,…
118 #ifdef DEBUG…
122 #endif…
125 #define ENUM_ITEM(T)            \…
129 #undef ENUM_ITEM…
131 }
132
1 call | 1 ref
133 void CSAGenerator::EmitInstruction(const PeekInstruction& instruction,…
136 }
137
1 call | 1 ref
138 void CSAGenerator::EmitInstruction(const PokeInstruction& instruction,…
142 }
143
1 call | 1 ref
144 void CSAGenerator::EmitInstruction(const DeleteRangeInstruction& instruction,…
147 }
148
1 call
149 void CSAGenerator::EmitInstruction(…
```

Init macro

```
17  base::Optional<Stack<std::string>> CSAGenerator::EmitGraph(
        3 refs
18      Stack<std::string> parameters) {
19    for (BottomOffset i = 0; i < parameters.AboveTop(); ++i) {
20      SetDefinitionVariable(DefinitionLocation::Parameter(i.offset),
21                            parameters.Peek(i));
22    }
23
24    for (Block* block : cfg_.blocks()) {
25      if (block->IsDead()) continue;
26
27      out() << "  compiler::CodeAssemblerParameterizedLabel<";
28      bool first = true;
29      DCHECK_EQ(block->InputTypes().Size(), block->InputDefinitions().Size());
30      for (BottomOffset i = 0; i < block->InputTypes().AboveTop(); ++i) {
31        if (block->InputDefinitions().Peek(i).IsPhiFromBlock(block)) {
32          if (!first) out() << ", ";
33          out() << block->InputTypes().Peek(i)->GetGeneratedTNodeTypeName();
34          first = false;
35        }
36      }
37      out() << "> " << BlockName(block) << "(&ca_, compiler::CodeAssemblerLabel::"
38            << (block->IsDeferred() ? "kDeferred" : "kNonDeferred") << ");\n";
39    }
40
41    EmitInstruction(GotoInstruction{cfg_.start()}, &parameters);
42    for (Block* block : cfg_.blocks()) {
43      if (cfg_.end() && *cfg_.end() == block) continue;
44      if (block->IsDead()) continue;
45      out() << "\n";
46
47      // Redirect the output of non-declarations into a buffer and only output
48      // declarations right away.
49      std::stringstream out_buffer;
50      std::ostream* old_out = out_;
51      out_ = &out_buffer;
52
53      out() << "  if (" << BlockName(block) << ".is_used()) {\n";
54      EmitBlock(block);
55      out() << "  }\n";
56
57      // All declarations have been printed now, so we can append the buffered
58      // output and redirect back to the original output stream.
59      out_ = old_out;
60      out() << out_buffer.str();
61    }
62    if (cfg_.end()) {
63      out() << "\n";
64      return EmitBlock(*cfg_.end());
65    }
66    return base::nullopt;
67  }
68
```

```
208  namespace internal {
209
       1 call | 36 refs | 9 refs | 1 ref | 4 refs
210  TNode<Number> GenericArrayUnshift_0(compiler::CodeAssemblerState* state_, TNode<Context> p_context, TNo
211    compiler::CodeAssembler ca_(state_);
212    compiler::CodeAssemblerParameterizedLabel<> block0(&ca_, compiler::CodeAssemblerLabel::kNonDeferred);
213    compiler::CodeAssemblerParameterizedLabel<> block2(&ca_, compiler::CodeAssemblerLabel::kNonDeferred);
214    compiler::CodeAssemblerParameterizedLabel<> block4(&ca_, compiler::CodeAssemblerLabel::kNonDeferred);
215    compiler::CodeAssemblerParameterizedLabel<> block5(&ca_, compiler::CodeAssemblerLabel::kNonDeferred);
216    compiler::CodeAssemblerParameterizedLabel<Number> block8(&ca_, compiler::CodeAssemblerLabel::kNonDefe
217    compiler::CodeAssemblerParameterizedLabel<Number> block6(&ca_, compiler::CodeAssemblerLabel::kNonDefe
218    compiler::CodeAssemblerParameterizedLabel<Number> block9(&ca_, compiler::CodeAssemblerLabel::kNonDefe
219    compiler::CodeAssemblerParameterizedLabel<Number> block10(&ca_, compiler::CodeAssemblerLabel::kNonDef
220    compiler::CodeAssemblerParameterizedLabel<Number> block11(&ca_, compiler::CodeAssemblerLabel::kNonDef
221    compiler::CodeAssemblerParameterizedLabel<Number> block7(&ca_, compiler::CodeAssemblerLabel::kNonDefe
222    compiler::CodeAssemblerParameterizedLabel<Number, Smi> block14(&ca_, compiler::CodeAssemblerLabel::kN
223    compiler::CodeAssemblerParameterizedLabel<Number, Smi> block12(&ca_, compiler::CodeAssemblerLabel::kN
224    compiler::CodeAssemblerParameterizedLabel<Number, Smi> block13(&ca_, compiler::CodeAssemblerLabel::kN
225    compiler::CodeAssemblerParameterizedLabel<> block3(&ca_, compiler::CodeAssemblerLabel::kNonDeferred);
226    compiler::CodeAssemblerParameterizedLabel<> block15(&ca_, compiler::CodeAssemblerLabel::kNonDeferred)
227      ca_.Goto(&block0);
228
229    TNode<JSReceiver> tmp0;
230    TNode<Number> tmp1;
231    TNode<Smi> tmp2;
232    TNode<Smi> tmp3;
233    TNode<BoolT> tmp4;
234    if (block0.is_used()) {
235      ca_.Bind(&block0);
236      tmp0 = CodeStubAssembler(state_).ToObject_Inline(TNode<Context>{p_context}, TNode<Object>{p_receive
237      tmp1 = GetLengthProperty_0(state_, TNode<Context>{p_context}, TNode<Object>{tmp0});
238      tmp2 = Convert_Smi_intptr_0(state_, TNode<IntPtrT>{p_arguments.length});
239      tmp3 = FromConstexpr_Smi_constexpr_int31_0(state_, 0);
240      tmp4 = CodeStubAssembler(state_).SmiGreaterThan(TNode<Smi>{tmp2}, TNode<Smi>{tmp3});
241      ca_.Branch(tmp4, &block2, std::vector<Node*>{}, &block3, std::vector<Node*>{});
242    }
243
244    TNode<Number> tmp5;
245    TNode<Number> tmp6;
246    TNode<BoolT> tmp7;
247    if (block2.is_used()) {
248      ca_.Bind(&block2);
249      tmp5 = CodeStubAssembler(state_).NumberAdd(TNode<Number>{tmp1}, TNode<Number>{tmp2});
250      tmp6 = FromConstexpr_Number_constexpr_float64_0(state_, kMaxSafeInteger);
251      tmp7 = NumberIsGreaterThan_0(state_, TNode<Number>{tmp5}, TNode<Number>{tmp6});
252      ca_.Branch(tmp7, &block4, std::vector<Node*>{}, &block5, std::vector<Node*>{});
253    }
254
255    if (block4.is_used()) {
256      ca_.Bind(&block4);
257      CodeStubAssembler(state_).ThrowTypeError(TNode<Context>{p_context}, MessageTemplate::kInvalidArrayL
258    }
259
```

Block body emit

```
2 calls | 6 refs
Stack<std::string> CSAGenerator::EmitBlock(const Block* block) {
  Stack<std::string> stack;
  std::stringstream phi_names;

  for (BottomOffset i = 0; i < block->InputTypes().AboveTop(); ++i) {
    const auto& def = block->InputDefinitions().Peek(i);
    stack.Push(DefinitionToVariable(def));
    if (def.IsPhiFromBlock(block)) {
      decls() << "  TNode<"
              << block->InputTypes().Peek(i)->GetGeneratedTNodeTypeName()
              << "> " << stack.Top() << ";\n";
      phi_names << ", &" << stack.Top();
    }
  }
  out() << "    ca_.Bind(&" << BlockName(block) << phi_names.str() << ");\n";

  for (const Instruction& instruction : block->instructions()) {
    EmitInstruction(instruction, &stack);
  }
  return stack;
}
```

```
232   TNode<Smi> tmp3;
233   TNode<BoolT> tmp4;
234   if (block0.is_used()) {
235     ca_.Bind(&block0);
236     tmp0 = CodeStubAssembler(state_).ToObject_Inline(TNode<Context>{p_context}, TNode<Object>{p_receiver}
237     tmp1 = GetLengthProperty_0(state_, TNode<Context>{p_context}, TNode<Object>{tmp0});
238     tmp2 = Convert_Smi_intptr_0(state_, TNode<IntPtrT>{p_arguments.length});
239     tmp3 = FromConstexpr_Smi_constexpr_int31_0(state_, 0);
240     tmp4 = CodeStubAssembler(state_).SmiGreaterThan(TNode<Smi>{tmp2}, TNode<Smi>{tmp3});
241     ca_.Branch(tmp4, &block2, std::vector<Node*>{}, &block3, std::vector<Node*>{});
242   }
243
244   TNode<Number> tmp5;
245   TNode<Number> tmp6;
246   TNode<BoolT> tmp7;
247   if (block2.is_used()) {
248     ca_.Bind(&block2);
249     tmp5 = CodeStubAssembler(state_).NumberAdd(TNode<Number>{tmp1}, TNode<Number>{tmp2});
250     tmp6 = FromConstexpr_Number_constexpr_float64_0(state_, kMaxSafeInteger);
251     tmp7 = NumberIsGreaterThan_0(state_, TNode<Number>{tmp5}, TNode<Number>{tmp6});
252     ca_.Branch(tmp7, &block4, std::vector<Node*>{}, &block5, std::vector<Node*>{});
253   }
254
```

Call csa macro emit

```
367   if (ExternMacro* extern_macro = ExternMacro::DynamicCast(instruction.macro)) {
368     out() << extern_macro->external_assembler_name() << "(state_).";
369   } else {
370     args.insert(args.begin(), "state_");
371   }
372   out() << instruction.macro->ExternalName() << "(";
373   PrintCommaSeparatedList(out(), args);
374   if (needs_flattening) {
375     out() << ").Flatten();\n";
376   } else {
377     out() << ");\n";
378   }
```

```
TNode<JSReceiver> tmp0;
TNode<Number> tmp1;
TNode<Smi> tmp2;
TNode<Smi> tmp3;
TNode<BoolT> tmp4;
if (block0.is_used()) {
  ca_.Bind(&block0);
  tmp0 = CodeStubAssembler(state_).ToObject_Inline(TNode<Context>{p_context}, TNode<Object>{p_receiv
  tmp1 = GetLengthProperty_0(state_, TNode<Context>{p_context}, TNode<Object>{tmp0});
  tmp2 = Convert_Smi_intptr_0(state_, TNode<IntPtrT>{p_arguments.length});
  tmp3 = FromConstexpr_Smi_constexpr_int31_0(state_, 0);
  tmp4 = CodeStubAssembler(state_).SmiGreaterThan(TNode<Smi>{tmp2}, TNode<Smi>{tmp3});
  ca_.Branch(tmp4, &block2, std::vector<Node*>{}, &block3, std::vector<Node*>{});
}
```