ISCAS 中国科学院软件研究所 Institute of Software Chinese Academy of Sciences | 智能软件研究中心 Intelligent Software Research Center

V8 Turbofan:
从字节码到SON图-part2

PLCT实验室 邱吉
qiuji@iscas.ac.cn

2022/03/09

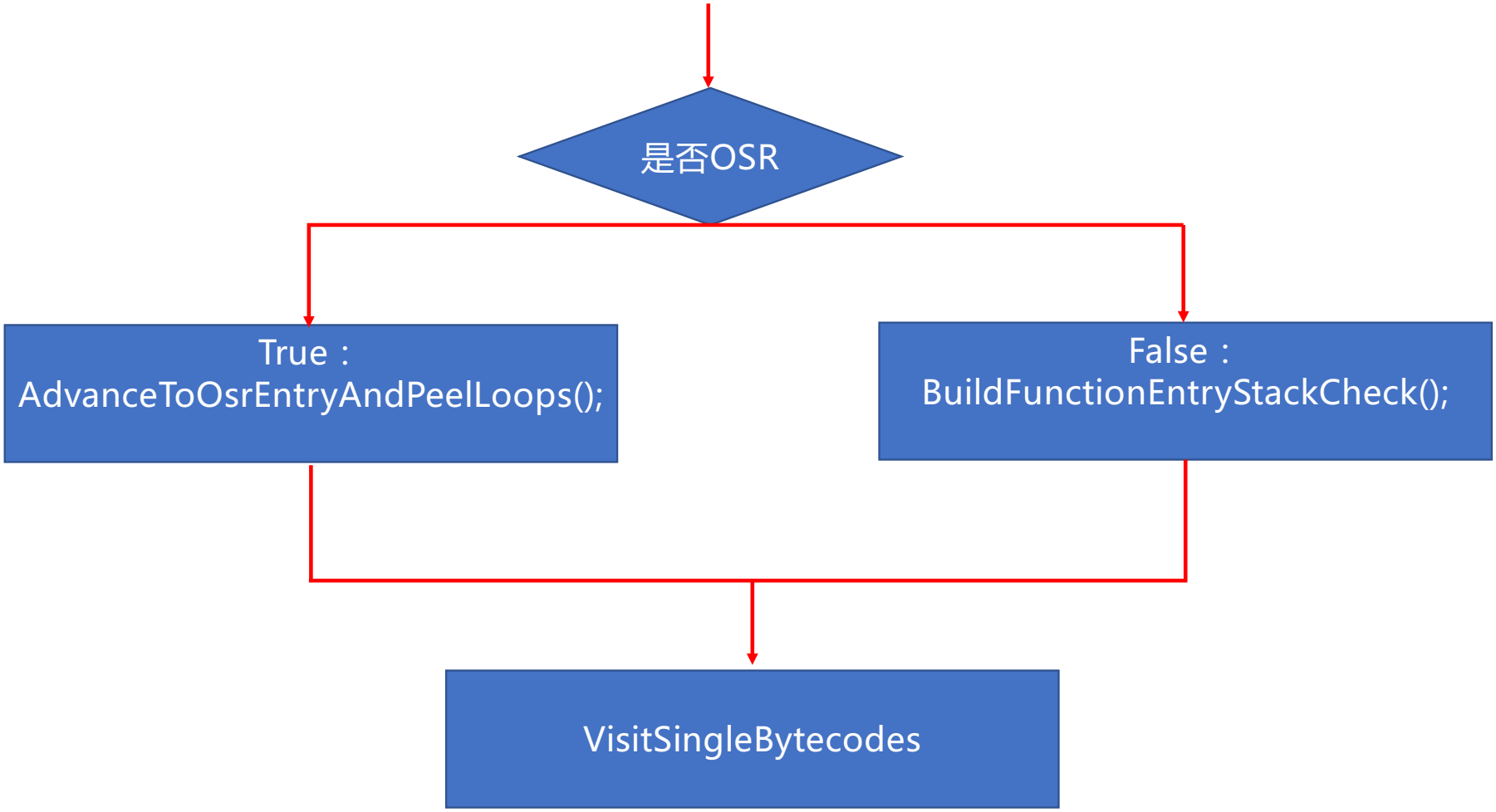
从Bytecode到SON图的构建：内容大纲

- Demo case 及其 Graph
- 总体构建流程概述
- Step by Step, Node by Node讲述图的构建过程
    1. new and set the Start node
    2. new Environment and set_environment
    3. CreateFeedbackCellNode
    4. CreateFeedbackVectorNode
    5. MaybeBuildTierUpCheck
    6. CreateNativeContextNode
    7. VisitBytecodes: one by one
    8. new and set the End node

Step by Step, Node by Node讲述图的构建过程

1. new and set the Start node
2. new Environment and set_environment
3. CreateFeedbackCellNode
4. CreateFeedbackVectorNode
5. MaybeBuildTierUpCheck
6. CreateNativeContextNode
7. VisitBytecodes: one by one
8. new and set the End node

Step7：VisitBytecodes整体流程



Step7：VisitBytecodes整体代码

@src/compiler/bytecode-graph-builder.cc

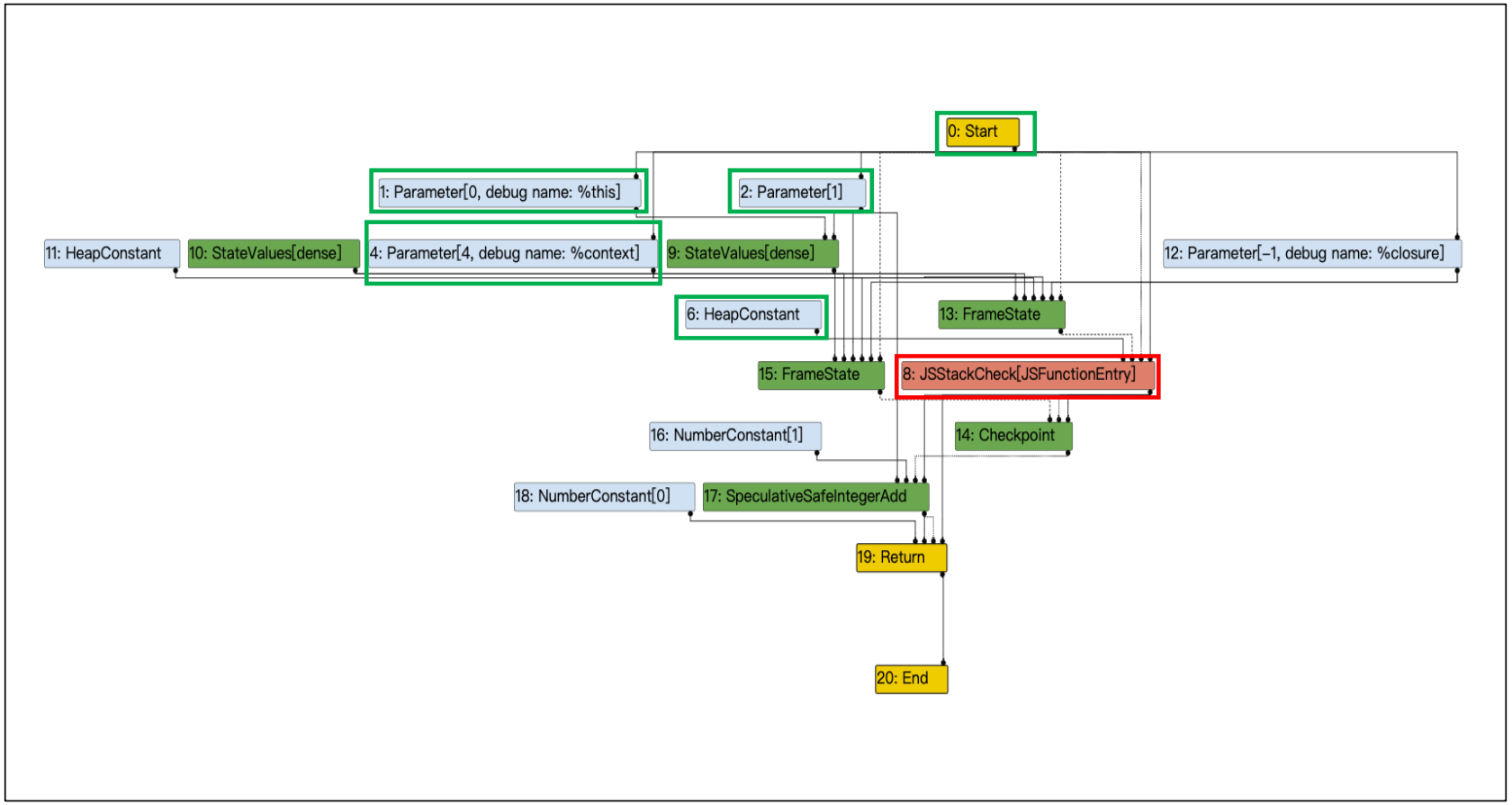
```
void BytecodeGraphBuilder::VisitBytecodes() {
  if (osr_) {
    // We peel the OSR loop and any outer loop containing it except that we
    // leave the nodes corresponding to the whole outermost loop (including
    // the last copies of the loops it contains) to be generated by the normal
    // bytecode iteration below.
    AdvanceToOsrEntryAndPeelLoops();
  } else {
    BuildFunctionEntryStackCheck(); //Step7.1： test函数要整体被JIT，不属于OSR
  }
  for (; !bytecode_iterator().done(); bytecode_iterator().Advance()) {
    VisitSingleBytecode(); //Build每一个Bytecode
  }
}
```

Step7.1: BuildFunctionEntryStackCheck

@src/compiler/bytecode-graph-builder.cc

```
void BytecodeGraphBuilder::BuildFunctionEntryStackCheck() {
  if (!skip_first_stack_check()) {
    DCHECK(exception_handlers_.empty());
    Node* node =
        NewNode(javascript()->StackCheck(StackCheckKind::kJSFunctionEntry)); //生成StackCheck节点，
StackCheck节点有一个输入是FrameState，现在先把FrameState输入设置成一个Dead节点
    PrepareFrameStateForFunctionEntryStackCheck(node);
  }
}
```

已构建完成的节点


0: Start
1: Parameter[0, debug name: %this]
2: Parameter[1]
11: HeapConstant
10: StateValues[dense]
4: Parameter[4, debug name: %context]
9: StateValues[dense]
12: Parameter[-1, debug name: %closure]
6: HeapConstant
13: FrameState
15: FrameState
8: JSStackCheck[JSFunctionEntry]
16: NumberConstant[1]
14: Checkpoint
18: NumberConstant[0]
17: SpeculativeSafeIntegerAdd
19: Return
20: End

节点列表

0	Start	
1	Parameter	*this
2	Parameter	1
3	HeapConstant	UndefinedConstant（用于values_向量中的register和accumulator初始值）
4	Parameter	context
5	HeapConstant	指向FeedBackVector
6	HeapConstant	nativecontext
7	Dead	JSStackCheck的一个Dummy输入
8	JSStackCheck	

Step7.2: PrepareFrameStateForFunctionEntryStackCheck

@src/compiler/bytecode-graph-builder.cc

```
void PrepareFrameStateForFunctionEntryStackCheck(Node* node) {
    return PrepareFrameState(node, OutputFrameStateCombine::Ignore(),
                             BytecodeOffset(kFunctionEntryBytecodeOffset),
                             bytecode_analysis().GetInLivenessFor(0));
  }
```

@src/compiler/bytecode-graph-builder.cc

```
void BytecodeGraphBuilder::PrepareFrameState(
    Node* node, OutputFrameStateCombine combine, BytecodeOffset bailout_id,
    const BytecodeLivenessState* liveness) {
  if (OperatorProperties::HasFrameStateInput(node->op())) {
    Node* frame_state_after = environment()->Checkpoint(bailout_id, combine, liveness); //Step7.3
    NodeProperties::ReplaceFrameStateInput(node, frame_state_after);
  }
}
```

Step7.3: environment()->Checkpoint

@src/compiler/bytecode-graph-builder.cc

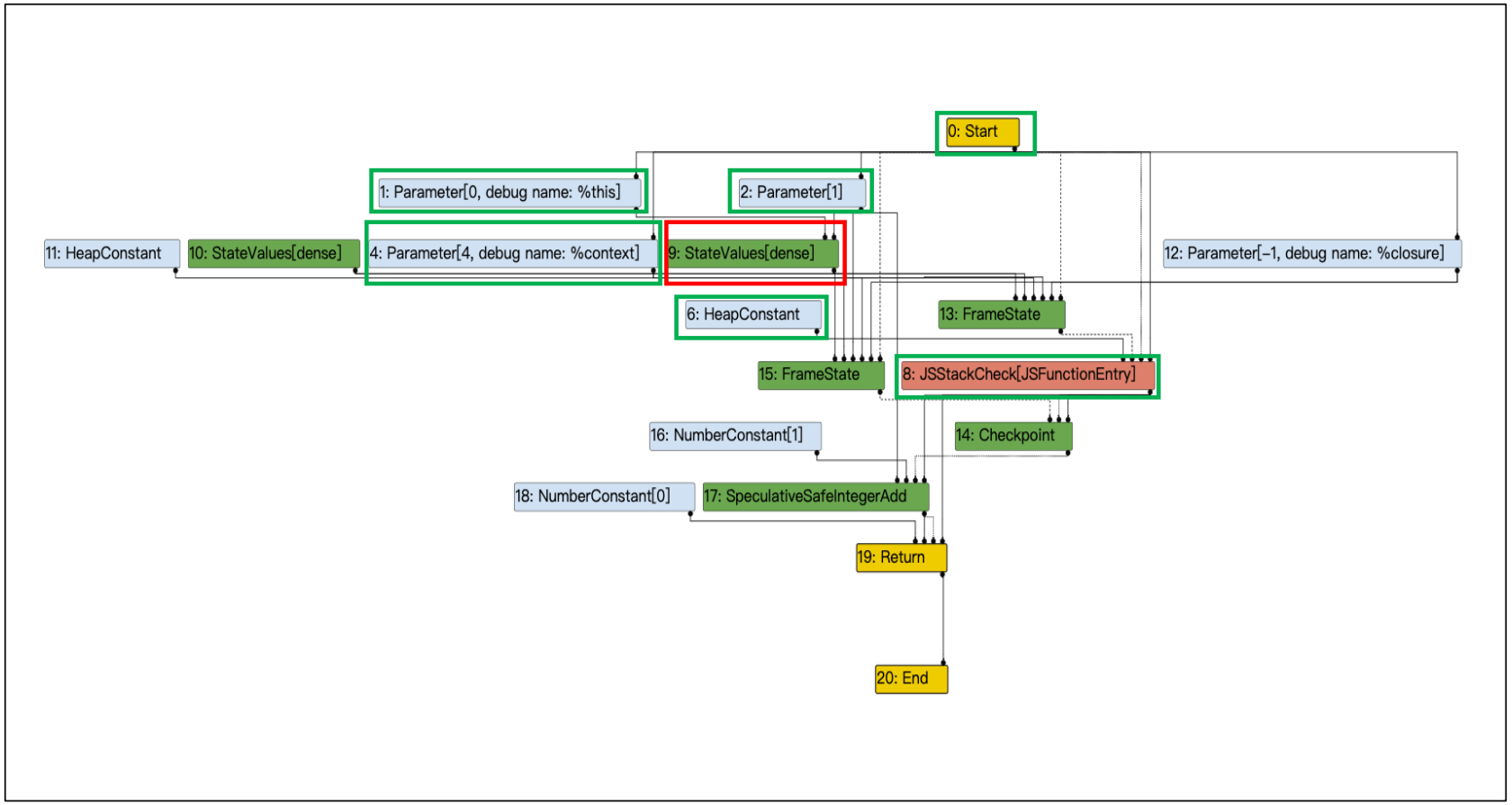
```
Node* BytecodeGraphBuilder::Environment::Checkpoint(
BytecodeOffset bailout_id, OutputFrameStateCombine combine, const BytecodeLivenessState* liveness) {
   UpdateStateValues(&parameters_state_values_, &values()->at(0), parameter_count()) ; //Step7.4: FrameState的第一个输入：parameters_state_values
   Node* registers_state_values = //FrameState的第二个输入：registers_state_values
     GetStateValuesFromCache(&values()->at(register_base()), register_count(),
                             liveness ? &liveness->bit_vector() : nullptr, 0);

   bool accumulator_is_live = !liveness || liveness->AccumulatorIsLive();
   Node* accumulator_state_value = //FrameState的第三个输入：accumulator_state_value
     accumulator_is_live && combine != OutputFrameStateCombine::PokeAt(0)
         ? values()->at(accumulator_base())
         : builder()->jsgraph()->OptimizedOutConstant();

   const Operator* op = common()->FrameState(
      bailout_id, combine, builder()->frame_state_function_info());
   Node* result = graph()->NewNode( //新建真实的FrameState节点
       op, parameters_state_values_, registers_state_values,
       accumulator_state_value, Context(), builder()->GetFunctionClosure(),
       builder()->graph()->start());
   return result;
}
```

Step7.4: parameter_state_values: UpdateStateValues

@src/compiler/bytecode-graph-builder.cc

```
void BytecodeGraphBuilder::Environment::UpdateStateValues(Node** state_values,
                                                          Node** values, int count) {
  if (StateValuesRequireUpdate(state_values, values, count)) {
    const Operator* op = common()->StateValues(count, SparseInputMask::Dense());
    (*state_values) = graph()->NewNode(op, count, values); //新建StateValues for parameters
  }
}
```

已构建完成的节点


0: Start
1: Parameter[0, debug name: %this]
2: Parameter[1]
11: HeapConstant
10: StateValues[dense]
4: Parameter[4, debug name: %context]
9: StateValues[dense]
12: Parameter[-1, debug name: %closure]
6: HeapConstant
13: FrameState
15: FrameState
8: JSStackCheck[JSFunctionEntry]
16: NumberConstant[1]
14: Checkpoint
18: NumberConstant[0]
17: SpeculativeSafeIntegerAdd
19: Return
20: End

节点列表

0	Start	
1	Parameter	*this
2	Parameter	1
3	HeapConstant	UndefinedConstant（用于values_向量中的register和accumulator初始值）
4	Parameter	context
5	HeapConstant	指向FeedBackVector
6	HeapConstant	nativecontext
7	Dead	JSStackCheck的一个Dummy输入
8	JSStackCheck	
9	StateValues	StateValues for parameter state, 作为FrameState的第一个输入

Step7.5: registers_state_values: GetStateValuesFromCache

@src/compiler/bytecode-graph-builder.cc： BytecodeGraphBuilder::Environment::Checkpoint()

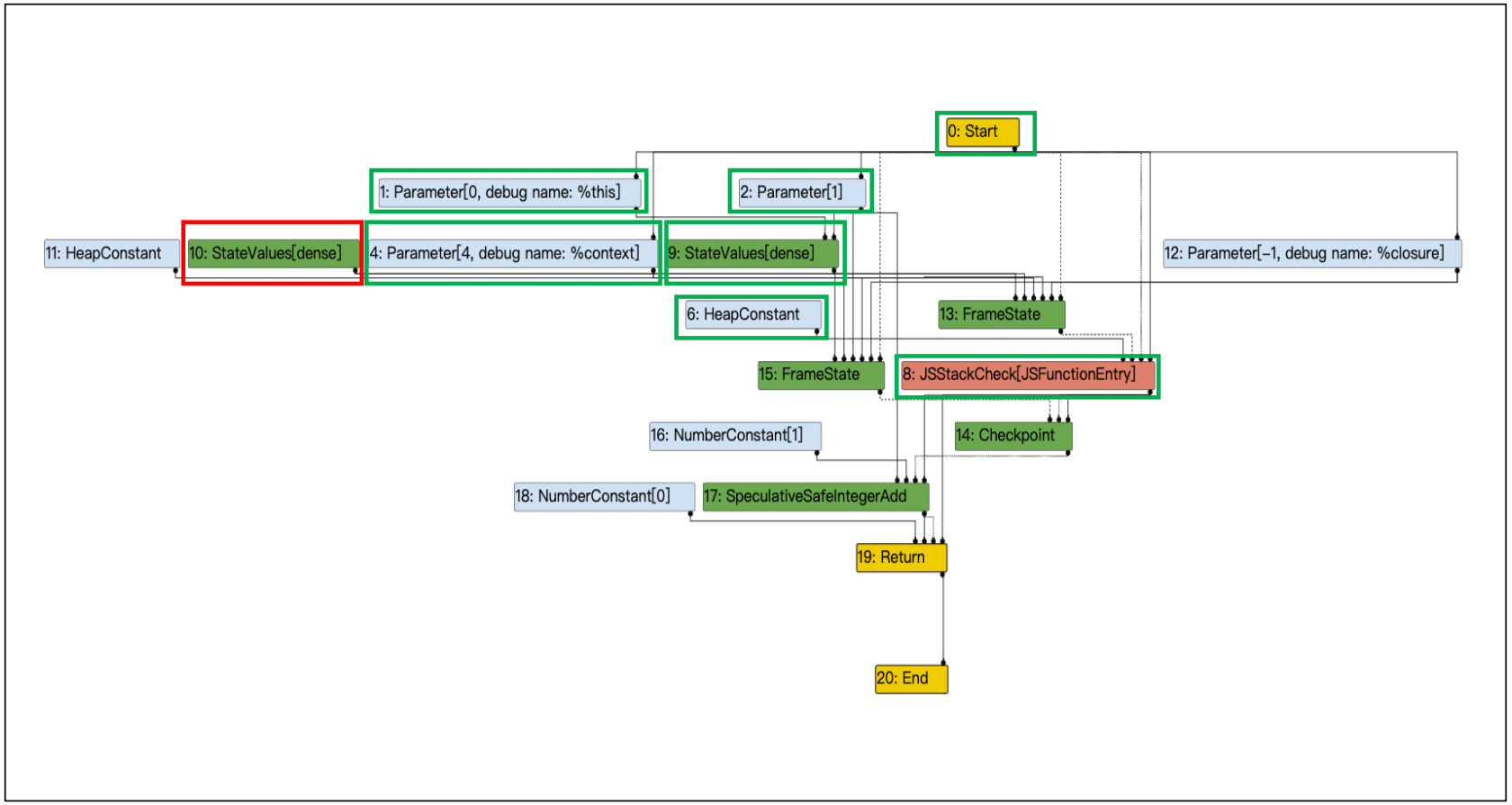
```
Node* registers_state_values = //FrameState的第二个输入：registers_state_values
    GetStateValuesFromCache(&values()->at(register_base()), register_count(),
                            liveness ? &liveness->bit_vector() : nullptr, 0);
```

```
Node* BytecodeGraphBuilder::Environment::GetStateValuesFromCache(
    Node** values, int count, const BitVector* liveness, int liveness_offset) {
  return builder_->state_values_cache_.GetNodeForValues(
      values, static_cast<size_t>(count), liveness, liveness_offset);
}
```

Step7.5: registers_state_values: GetStateValuesFromCache

```
@ src/compiler/state-values-utils.cc

Node* StateValuesCache::GetNodeForValues(Node** values, size_t count,
                                         const BitVector* liveness,
                                         int liveness_offset) {
  if (count == 0) {
    return GetEmptyStateValues(); //test函数的register 个数是0
  }
  size_t height = 0;
  size_t max_inputs = kMaxInputCount;
  while (count > max_inputs) {
    height++;
    max_inputs *= kMaxInputCount;
  }
  size_t values_idx = 0;
  Node* tree = BuildTree(&values_idx, values, count, liveness, liveness_offset, height); //构建树状结构的
节点cache
  return tree;
}
```

已构建完成的节点


0: Start
1: Parameter[0, debug name: %this]
2: Parameter[1]
11: HeapConstant
10: StateValues[dense]
4: Parameter[4, debug name: %context]
9: StateValues[dense]
12: Parameter[-1, debug name: %closure]
6: HeapConstant
13: FrameState
15: FrameState
8: JSStackCheck[JSFunctionEntry]
16: NumberConstant[1]
14: Checkpoint
18: NumberConstant[0]
17: SpeculativeSafeIntegerAdd
19: Return
20: End

节点列表

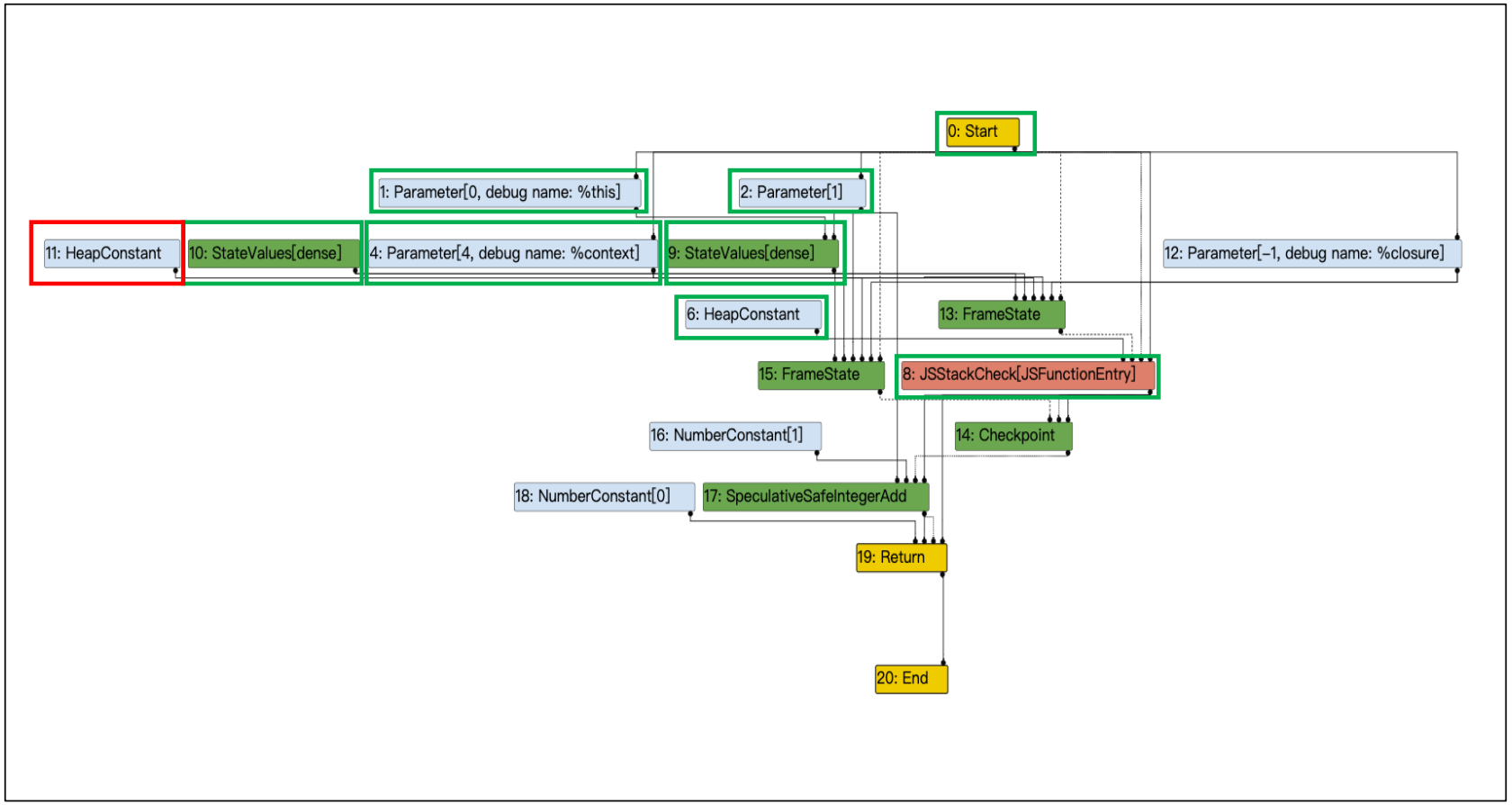
0	Start	
1	Parameter	*this
2	Parameter	1
3	HeapConstant	UndefinedConstant（用于values_向量中的register和accumulator初始值）
4	Parameter	context
5	HeapConstant	指向FeedBackVector
6	HeapConstant	nativecontext
7	Dead	JSStackCheck的一个Dummy输入
8	JSStackCheck	
9	StateValues	StateValues for parameter state, 作为FrameState的第一个输入
10	StateValues	StateValues for register states, 作为FrameState的第二个输入

Step7.6: accumulator_state_value

@src/compiler/bytecode-graph-builder.cc： BytecodeGraphBuilder::Environment::Checkpoint()

```
    bool accumulator_is_live = !liveness || liveness->AccumulatorIsLive();
    Node* accumulator_state_value = //FrameState的第三个输入：accumulator_state_value
      accumulator_is_live && combine != OutputFrameStateCombine::PokeAt(0)
          ? values()->at(accumulator_base())
          : builder()->jsgraph()->OptimizedOutConstant(); //在函数入口处，accumulator liveness是false，所以直接生成一个HeapConstant节点作为accumulator的state
```

已构建完成的节点


0: Start
1: Parameter[0, debug name: %this]
2: Parameter[1]
11: HeapConstant
10: StateValues[dense]
4: Parameter[4, debug name: %context]
9: StateValues[dense]
12: Parameter[-1, debug name: %closure]
6: HeapConstant
13: FrameState
15: FrameState
8: JSStackCheck[JSFunctionEntry]
16: NumberConstant[1]
14: Checkpoint
18: NumberConstant[0]
17: SpeculativeSafeIntegerAdd
19: Return
20: End

节点列表

0	Start	
1	Parameter	*this
2	Parameter	1
3	HeapConstant	UndefinedConstant（用于values_向量中的register和accumulator初始值）
4	Parameter	context
5	HeapConstant	指向FeedBackVector
6	HeapConstant	nativecontext
7	Dead	JSStackCheck的一个Dummy输入
8	JSStackCheck	
9	StateValues	StateValues for parameter state, 作为FrameState的第一个输入
10	StateValues	StateValues for register states, 作为FrameState的第二个输入
11	HeapConstant	StateValues for accumulator states,作为FrameState的第三个输入

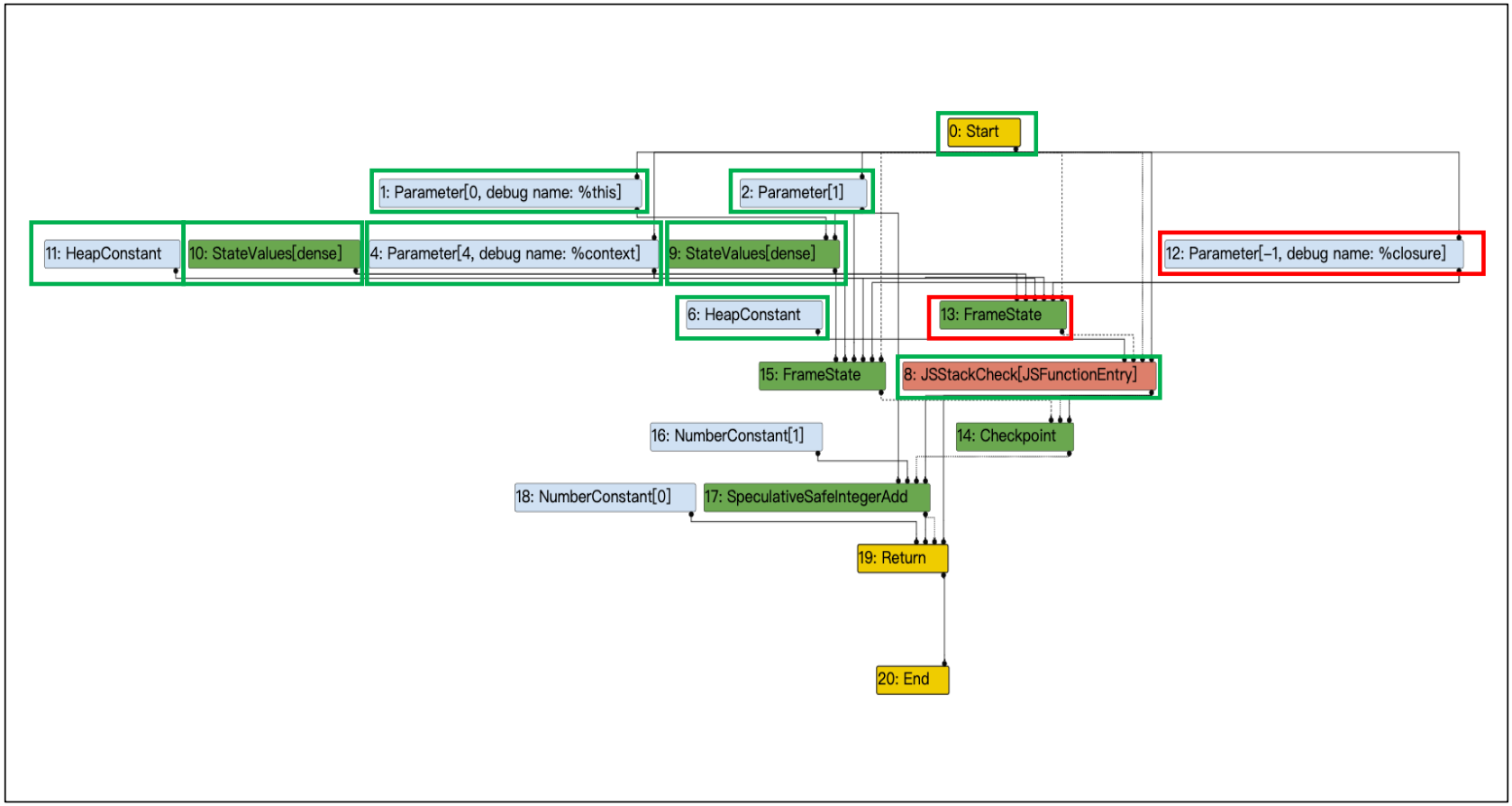
Step7.7: new FrameState node for StackCheck

@src/compiler/bytecode-graph-builder.cc： BytecodeGraphBuilder::Environment::Checkpoint()

```
const Operator* op = common()->FrameState(
    bailout_id, combine, builder()->frame_state_function_info());
Node* result = graph()->NewNode( //新建FrameState节点
    op, parameters_state_values_, registers_state_values,
    accumulator_state_value, Context(), builder()->GetFunctionClosure(), //新建FunctionClosure节点
    builder()->graph()->start());
return result;
```

```
Node* BytecodeGraphBuilder::GetFunctionClosure() {
  if (!function_closure_.is_set()) {
    int index = Linkage::kJSCallClosureParamIndex;
    Node* node = GetParameter(index, "%closure");
    function_closure_.set(node);
  }
  return function_closure_.get();
}
```

已构建完成的节点


0: Start
1: Parameter[0, debug name: %this]
2: Parameter[1]
11: HeapConstant
10: StateValues[dense]
4: Parameter[4, debug name: %context]
9: StateValues[dense]
12: Parameter[-1, debug name: %closure]
6: HeapConstant
13: FrameState
15: FrameState
8: JSStackCheck[JSFunctionEntry]
16: NumberConstant[1]
14: Checkpoint
18: NumberConstant[0]
17: SpeculativeSafeIntegerAdd
19: Return
20: End

节点列表

0	Start	
1	Parameter	*this
2	Parameter	1
3	HeapConstant	UndefinedConstant（用于values_向量中的register和accumulator初始值）
4	Parameter	context
5	HeapConstant	指向FeedBackVector
6	HeapConstant	nativecontext
7	Dead	JSStackCheck的一个Dummy输入
8	JSStackCheck	
9	StateValues	StateValues for parameter state, 作为FrameState的第一个输入
10	StateValues	StateValues for register states, 作为FrameState的第二个输入
11	HeapConstant	StateValues for accumulator states,作为FrameState的第三个输入
12	Parameter	StateValues for function closure states,作为FrameState的第四个输入
13	FrameState	替换JSStackCheck的一个Dummy输入

Step7.8: VisitSingleBytecode-整体流程

@src/compiler/bytecode-graph-builder.cc

```
void BytecodeGraphBuilder::VisitSingleBytecode() {
  tick_counter_->TickAndMaybeEnterSafepoint();
  int current_offset = bytecode_iterator().current_offset();
  UpdateSourcePosition(current_offset);
  ExitThenEnterExceptionHandlers(current_offset);
  DCHECK_GE(exception_handlers_.empty() ? current_offset
                                        : exception_handlers_.top().end_offset_,
            current_offset);
  SwitchToMergeEnvironment(current_offset);

  if (environment() != nullptr) {
    BuildLoopHeaderEnvironment(current_offset);

    switch (bytecode_iterator().current_bytecode()) { //switch-case构成分派，visitor模式
#define BYTECODE_CASE(name, ...)       \
  case interpreter::Bytecode::k##name: \
    Visit##name();                     \
    break;
      BYTECODE_LIST(BYTECODE_CASE)
#undef BYTECODE_CASE
    } } }
```

Step7.9: VisitLdar: Ldar a0（第一条字节码）

@src/compiler/bytecode-graph-builder.cc

```
void BytecodeGraphBuilder::VisitLdar() {
  Node* value =
      environment()->LookupRegister(bytecode_iterator().GetRegisterOperand(0)); //在environment的values_数组中找到a0寄存器对应的node
  environment()->BindAccumulator(value); //将environment中的values_数组中accumulator对应的node指针指向value
}
```

Step7.10: VisitAddSmi： AddSmi [1], [0]（第二条字节码）

@src/compiler/bytecode-graph-builder.cc

```
void BytecodeGraphBuilder::VisitAddSmi() {
  FeedbackSource feedback = CreateFeedbackSource(
      bytecode_iterator().GetSlotOperand(kBinaryOperationSmiHintIndex));
  BuildBinaryOpWithImmediate(javascript()->Add(feedback));
}
```

Step7.10: BuildBinaryOpWithImmediate

@src/compiler/bytecode-graph-builder.cc

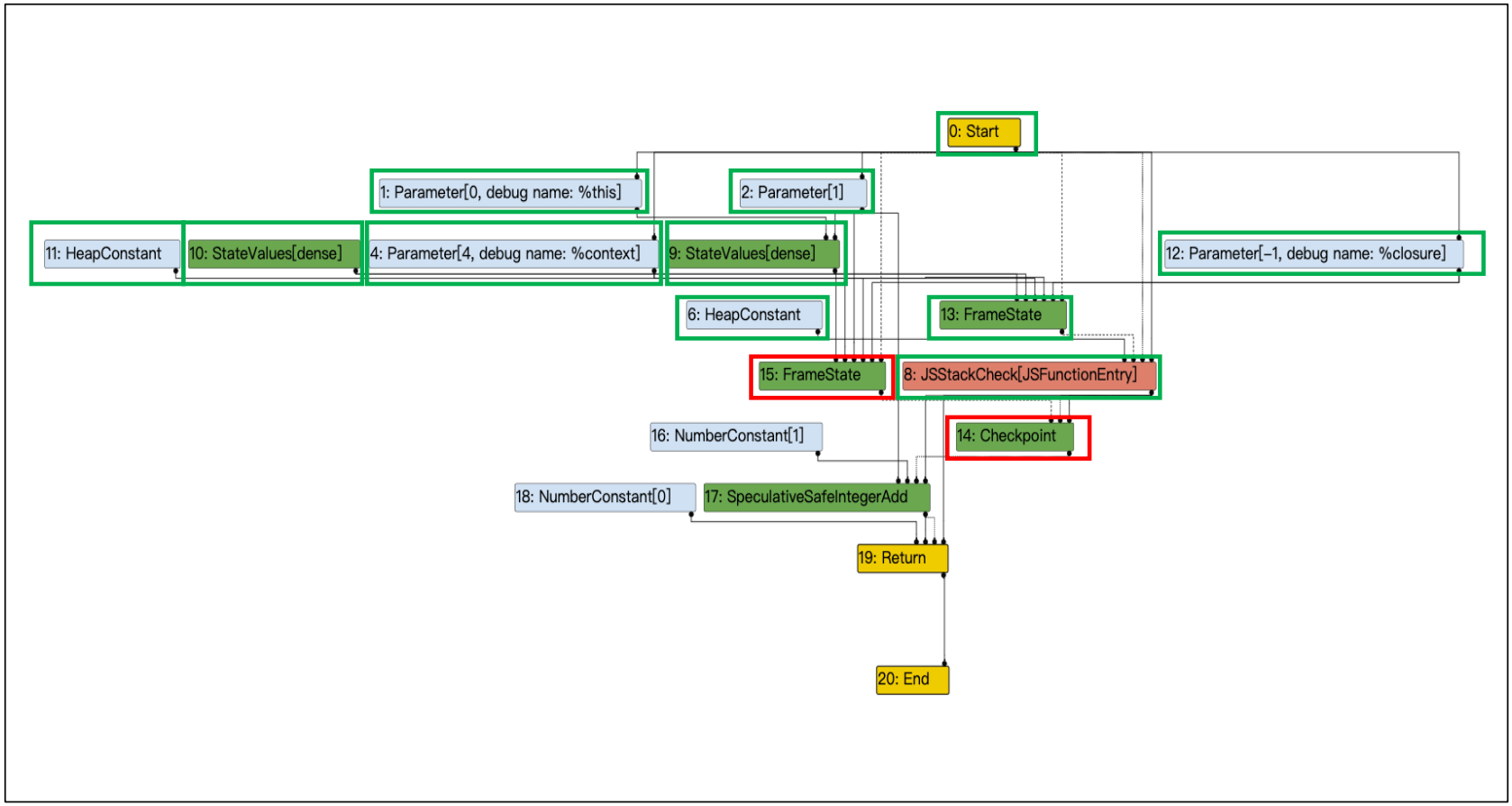
```
void BytecodeGraphBuilder::BuildBinaryOpWithImmediate(const Operator* op) {
  DCHECK(JSOperator::IsBinaryWithFeedback(op->opcode()));
  PrepareEagerCheckpoint(); //Step 7.11 准备生成激进的CheckPoint
  Node* left = environment()->LookupAccumulator(); //从env中找到accumulator对应的node节点
  Node* right = jsgraph()->Constant(bytecode_iterator().GetImmediateOperand(0)); //Step 7.12将右操作数的常量1，
生成一个constant节点
  FeedbackSlot slot =
      bytecode_iterator().GetSlotOperand(kBinaryOperationSmiHintIndex); //从[0]中获得slot
  JSTypeHintLowering::LoweringResult lowering =
      TryBuildSimplifiedBinaryOp(op, left, right, slot); //尝试构建SimplifiedBinaryOp
  if (lowering.IsExit()) return;
  Node* node = nullptr;
  if (lowering.IsSideEffectFree()) {
    node = lowering.value();
  } else {
    DCHECK(!lowering.Changed());
    DCHECK(IrOpcode::IsFeedbackCollectingOpcode(op->opcode()));
    node = NewNode(op, left, right, feedback_vector_node());
  }
  environment()->BindAccumulator(node, Environment::kAttachFrameState);
}
```

Step7.11: PrepareEagerCheckpoint

@src/compiler/bytecode-graph-builder.cc

```
void BytecodeGraphBuilder::PrepareEagerCheckpoint() {
  if (needs_eager_checkpoint()) {
    // Create an explicit checkpoint node for before the operation. This only
    // needs to happen if we aren't effect-dominated by a {Checkpoint} already.
    mark_as_needing_eager_checkpoint(false);
    Node* node = NewNode(common()->Checkpoint()); //创建一个checkpoint节点，跟StackCheck类似，它也需要一个Dummy的Dead FrameState节点，但之前已经有Dead节点了，所以不需要再次创建
    BytecodeOffset bailout_id(bytecode_iterator().current_offset()); //得到addsmi指令位置对应的bailout id
    const BytecodeLivenessState* liveness_before =
        bytecode_analysis().GetInLivenessFor(bytecode_iterator().current_offset()); //得到addsmi指令前的register liveness
    Node* frame_state_before = environment()->Checkpoint(
        bailout_id, OutputFrameStateCombine::Ignore(), liveness_before);  //从environment中查找这条指令前的checkpoint对应的FrameState，步骤同前述7.3～7.7过程
    NodeProperties::ReplaceFrameStateInput(node, frame_state_before); //将CheckPoint的Dead FrameState替换成AddSmi对应的FrameState节点
}
```

已构建完成的节点


0: Start
1: Parameter[0, debug name: %this]
2: Parameter[1]
11: HeapConstant
10: StateValues[dense]
4: Parameter[4, debug name: %context]
9: StateValues[dense]
12: Parameter[-1, debug name: %closure]
6: HeapConstant
13: FrameState
15: FrameState
8: JSStackCheck[JSFunctionEntry]
16: NumberConstant[1]
14: Checkpoint
18: NumberConstant[0]
17: SpeculativeSafeIntegerAdd
19: Return
20: End

节点列表

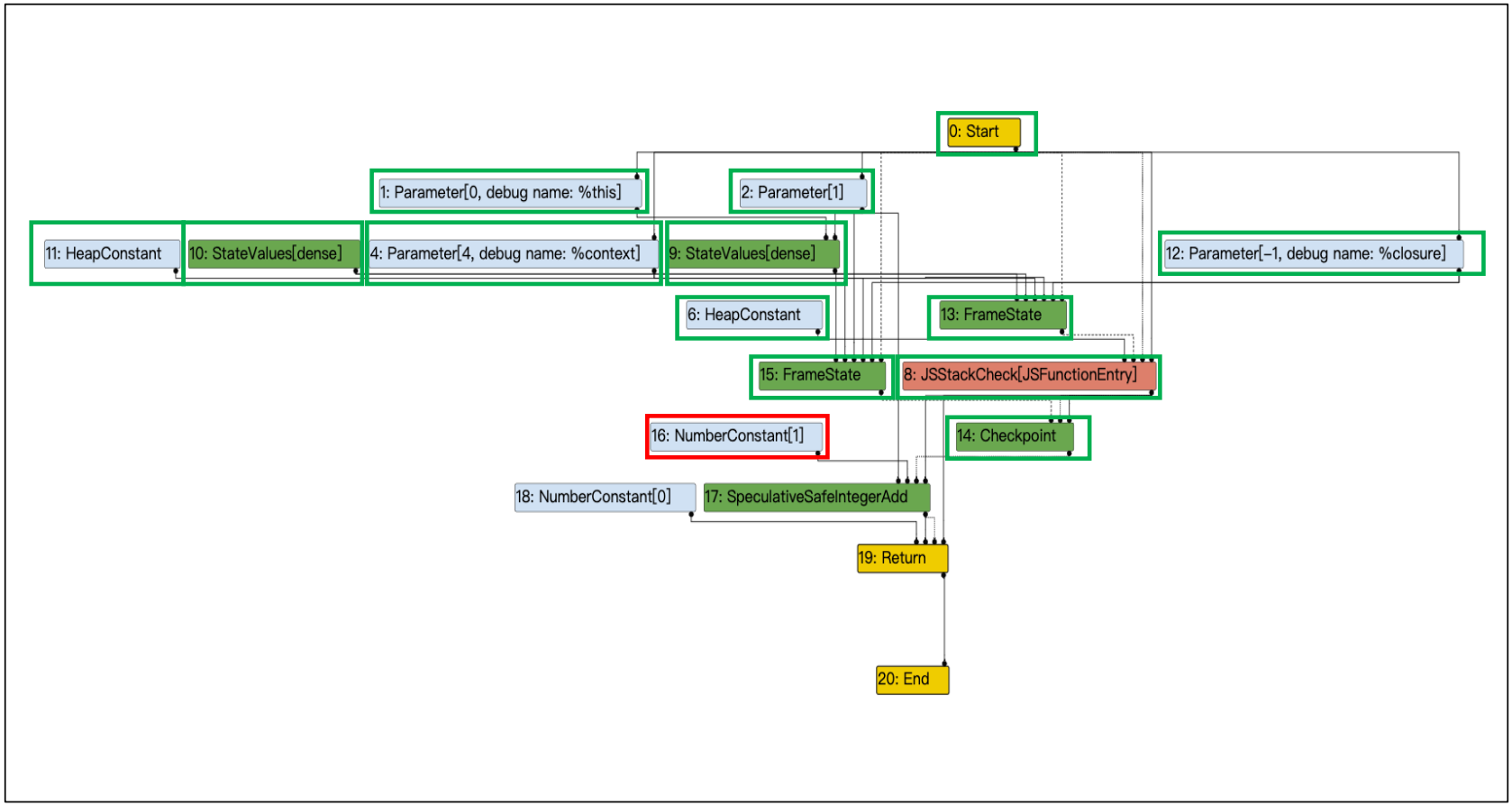
0	Start	
1	Parameter	*this
2	Parameter	1
3	HeapConstant	UndefinedConstant（用于values_向量中的register和accumulator初始值）
4	Parameter	context
5	HeapConstant	指向FeedBackVector
6	HeapConstant	nativecontext
7	Dead	JSStackCheck的一个Dummy输入
8	JSStackCheck	
9	StateValues	StateValues for parameter state, 作为FrameState的第一个输入
10	StateValues	StateValues for register states, 作为FrameState的第二个输入
11	HeapConstant	StateValues for accumulator states,作为FrameState的第三个输入
12	Parameter	StateValues for function closure states,作为FrameState的第四个输入
13	FrameState	替换JSStackCheck的一个Dummy输入
14	CheckPoint	作为AddSmi操作的Eager Checkpoint
15	FrameState	作为14号Checkpoint的FrameState输入

Step7.12: 为AddSmi的常量操作数1生成Constant节点

@src/compiler/bytecode-graph-builder.cc

```
void BytecodeGraphBuilder::BuildBinaryOpWithImmediate(const Operator* op) {
  DCHECK(JSOperator::IsBinaryWithFeedback(op->opcode()));
  PrepareEagerCheckpoint();
  Node* left = environment()->LookupAccumulator();
  Node* right = jsgraph()->Constant(bytecode_iterator().GetImmediateOperand(0)); //Step 7.12将右操作数的常量1，
生成一个constant节点
  FeedbackSlot slot =
      bytecode_iterator().GetSlotOperand(kBinaryOperationSmiHintIndex); //从[0]中获得slot
  JSTypeHintLowering::LoweringResult lowering =
      TryBuildSimplifiedBinaryOp(op, left, right, slot);
  if (lowering.IsExit()) return;
  Node* node = nullptr;
  if (lowering.IsSideEffectFree()) {
    node = lowering.value();
  } else {
    DCHECK(!lowering.Changed());
    DCHECK(IrOpcode::IsFeedbackCollectingOpcode(op->opcode()));
    node = NewNode(op, left, right, feedback_vector_node());
  }
  environment()->BindAccumulator(node, Environment::kAttachFrameState);
}
```

已构建完成的节点


0: Start
1: Parameter[0, debug name: %this]
2: Parameter[1]
11: HeapConstant
10: StateValues[dense]
4: Parameter[4, debug name: %context]
9: StateValues[dense]
12: Parameter[-1, debug name: %closure]
6: HeapConstant
13: FrameState
15: FrameState
8: JSStackCheck[JSFunctionEntry]
16: NumberConstant[1]
14: Checkpoint
18: NumberConstant[0]
17: SpeculativeSafeIntegerAdd
19: Return
20: End

节点列表

0	Start	
1	Parameter	*this
2	Parameter	1
3	HeapConstant	UndefinedConstant（用于values_向量中的register和accumulator初始值）
4	Parameter	context
5	HeapConstant	指向FeedBackVector
6	HeapConstant	nativecontext
7	Dead	JSStackCheck的一个Dummy输入
8	JSStackCheck	
9	StateValues	StateValues for parameter state, 作为FrameState的第一个输入
10	StateValues	StateValues for register states, 作为FrameState的第二个输入
11	HeapConstant	StateValues for accumulator states,作为FrameState的第三个输入
12	Parameter	StateValues for function closure states,作为FrameState的第四个输入
13	FrameState	替换JSStackCheck的一个Dummy输入
14	CheckPoint	作为AddSmi操作的Eager Checkpoint
15	FrameState	作为14号Checkpoint的FrameState输入
16	NumberConstant	AddSmi的常量操作数1

Step7.13: TrySimplifiedBinaryOp

@src/compiler/bytecode-graph-builder.cc

```
JSTypeHintLowering::LoweringResult
BytecodeGraphBuilder::TryBuildSimplifiedBinaryOp(const Operator* op, Node* left,
                                                 Node* right,
                                                 FeedbackSlot slot) {
  Node* effect = environment()->GetEffectDependency();
  Node* control = environment()->GetControlDependency();
  JSTypeHintLowering::LoweringResult result =
      type_hint_lowering().ReduceBinaryOperation(op, left, right, effect, control, slot); //Step7.14
ReduceBinaryOperation
  ApplyEarlyReduction(result); //更新environment的control和effect依赖状态
  return result;
}
```

- type_hint_lowing()函数返回JSTypeHintLowing类型的成员对象type_hint_lowering
- JSTypeHintLowing类用于在从Bytecode构建SON图的过程中，根据FeedbackSlot提供的类型反馈信息，直接创建speculative simplified operator（跳过了本来应该创建的较高语义层次的JS operator）

Step7.14: ReduceBinaryOperation

@src/compiler/js-type-hint-lowering.cc

```
JSTypeHintLowering::LoweringResult JSTypeHintLowering::ReduceBinaryOperation(
    const Operator* op, Node* left, Node* right, Node* effect, Node* control, FeedbackSlot slot) const {
  switch (op->opcode()) {
  ...many ops...
    case IrOpcode::kJSAdd: {
      if (Node* node = TryBuildSoftDeopt(
              slot, effect, control, DeoptimizeReason::kInsufficientTypeFeedbackForBinaryOperation)) {
        return LoweringResult::Exit(node); } // SoftDeopt意味着FB信息还不足够，需要SoftDeopt到解释器
      JSSpeculativeBinopBuilder b(this, op, left, right, effect, control, slot); //创建JSSpeculativeBinopBuilder对象
      if (Node* node = b.TryBuildNumberBinop()) { //Step 7.15 执行TryBuildNumberBinop
            return LoweringResult::SideEffectFree(node, node, control); // 如果可以正确地BuildNumberBinop，那么，这
样的计算节点是不会引起Deopt的，没有副作用，所有返回SideEffectFree类型的LoweringResult
      }
      if (op->opcode() == IrOpcode::kJSAdd || op->opcode() == IrOpcode::kJSSubtract) {
        if (Node* node = b.TryBuildBigIntBinop()) { //如果反馈信息不是SmallInt，则尝试BuildBigIntBinop
          return LoweringResult::SideEffectFree(node, node, control);
        }}
      break; }
```

Step7.15: TryBuildNumberBinop

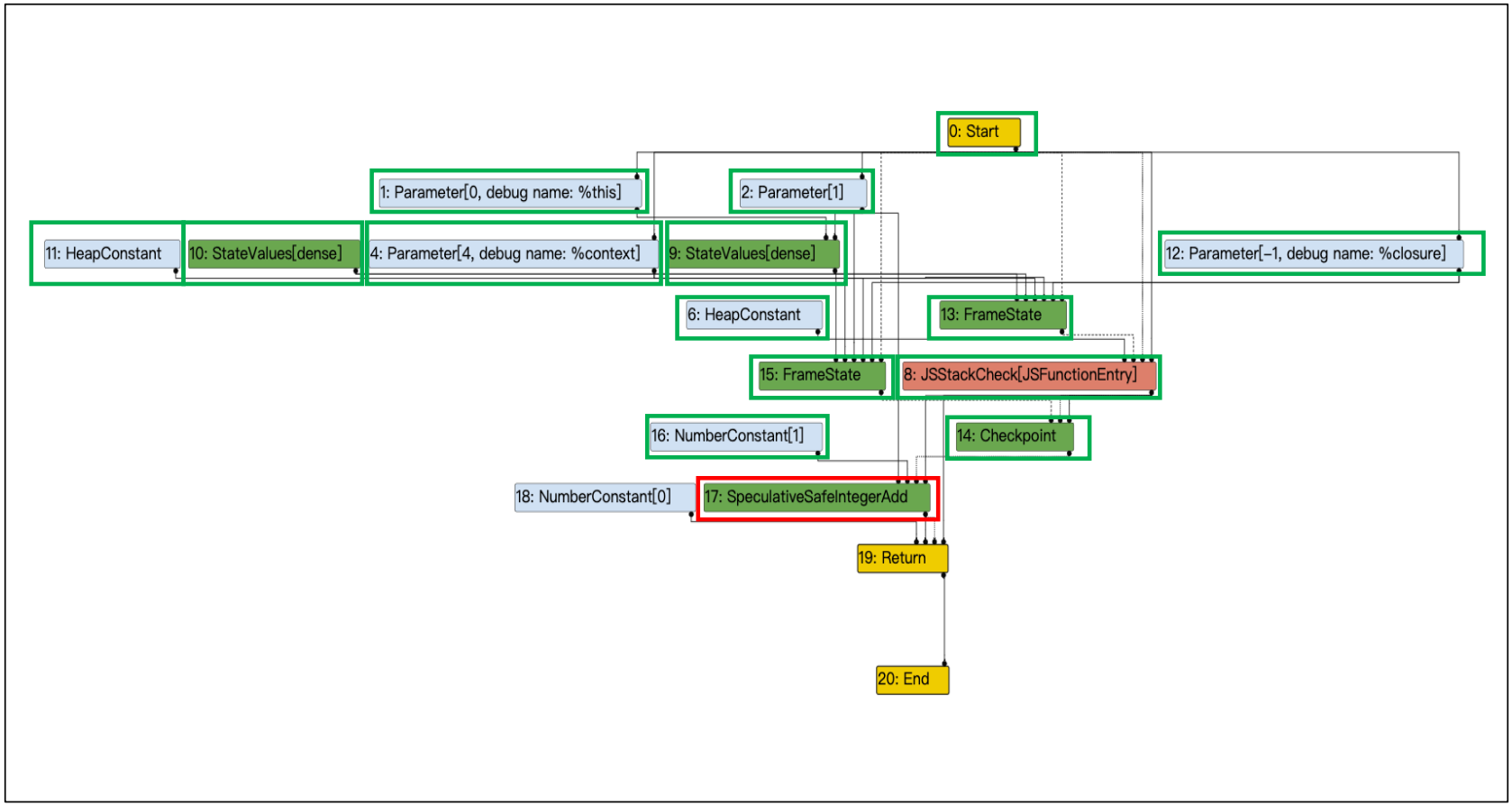
@src/compiler/js-type-hint-lowering.cc:

```
Node* TryBuildNumberBinop() {
    NumberOperationHint hint;
    if (GetBinaryNumberOperationHint(&hint)) { //从fb slot中得到type的hint判断它是不是BinaryNumber
      const Operator* op = SpeculativeNumberOp(hint); //得到SpeculativeSafeIntegerAdd opcode
      Node* node = BuildSpeculativeOperation(op); //最后调用得到SpeculativeSafeIntegerAdd的节点
      return node;
    }
    return nullptr;
  }
```

@src/compiler/js-type-hint-lowering.cc:

```
Node* BuildSpeculativeOperation(const Operator* op) {
    return graph()->NewNode(op, left_, right_, effect_, control_); //创建SpeculativeSafeIntegerAdd节点
  }
```

已构建完成的节点


0: Start
1: Parameter[0, debug name: %this]
2: Parameter[1]
11: HeapConstant
10: StateValues[dense]
4: Parameter[4, debug name: %context]
9: StateValues[dense]
12: Parameter[-1, debug name: %closure]
6: HeapConstant
13: FrameState
15: FrameState
8: JSStackCheck[JSFunctionEntry]
16: NumberConstant[1]
14: Checkpoint
18: NumberConstant[0]
17: SpeculativeSafeIntegerAdd
19: Return
20: End

节点列表

0	Start	
1	Parameter	*this
2	Parameter	1
3	HeapConstant	UndefinedConstant（用于values_向量中的register和accumulator初始值）
4	Parameter	context
5	HeapConstant	指向FeedBackVector
6	HeapConstant	nativecontext
7	Dead	JSStackCheck的一个Dummy输入
8	JSStackCheck	
9	StateValues	StateValues for parameter state, 作为FrameState的第一个输入
10	StateValues	StateValues for register states, 作为FrameState的第二个输入
11	HeapConstant	StateValues for accumulator states,作为FrameState的第三个输入
12	Parameter	StateValues for function closure states,作为FrameState的第四个输入
13	FrameState	替换JSStackCheck的一个Dummy输入
14	CheckPoint	作为AddSmi操作的Eager Checkpoint
15	FrameState	作为14号Checkpoint的FrameState输入
16	NumberConstant	AddSmi的常量操作数1
17	SpeculativeSafeIntegerAdd	加法的节点

Step7.16: Build Add操作扫尾：判断type_hint_lowering结果的属性

```
void BytecodeGraphBuilder::BuildBinaryOpWithImmediate(const Operator* op) {
  DCHECK(JSOperator::IsBinaryWithFeedback(op->opcode()));
  PrepareEagerCheckpoint(); //Step 7.11 准备生成激进的CheckPoint
  Node* left = environment()->LookupAccumulator(); //从env中找到accumulator对应的node节点
  Node* right = jsgraph()->Constant(bytecode_iterator().GetImmediateOperand(0)); //Step 7.12将右操作数的常量1，
生成一个constant节点
  FeedbackSlot slot =
      bytecode_iterator().GetSlotOperand(kBinaryOperationSmiHintIndex); //从[0]中获得slot
  JSTypeHintLowering::LoweringResult lowering =
      TryBuildSimplifiedBinaryOp(op, left, right, slot); //尝试构建SimplifiedBinaryOp
  if (lowering.IsExit()) return; // 返回的LoweringResult，如果Lowering的成Deopt节点了，就会导致退出JITed code，IsExit
就为true
  Node* node = nullptr;
  if (lowering.IsSideEffectFree()) {
    node = lowering.value(); //如果生成了Speculative的操作，那么SideEffect就是Free的
  } else {
    DCHECK(!lowering.Changed());
    DCHECK(IrOpcode::IsFeedbackCollectingOpcode(op->opcode()));
    node = NewNode(op, left, right, feedback_vector_node());
  }
  environment()->BindAccumulator(node, Environment::kAttachFrameState); //将node，也就是加法操作的结果绑定到
enviornment上去
}
```

*side effect free 是指不对图的其他部分产生额外的控制流和内存读写顺序的约束

Step7.17: Build Add操作扫尾：ApplyEarlyReduction

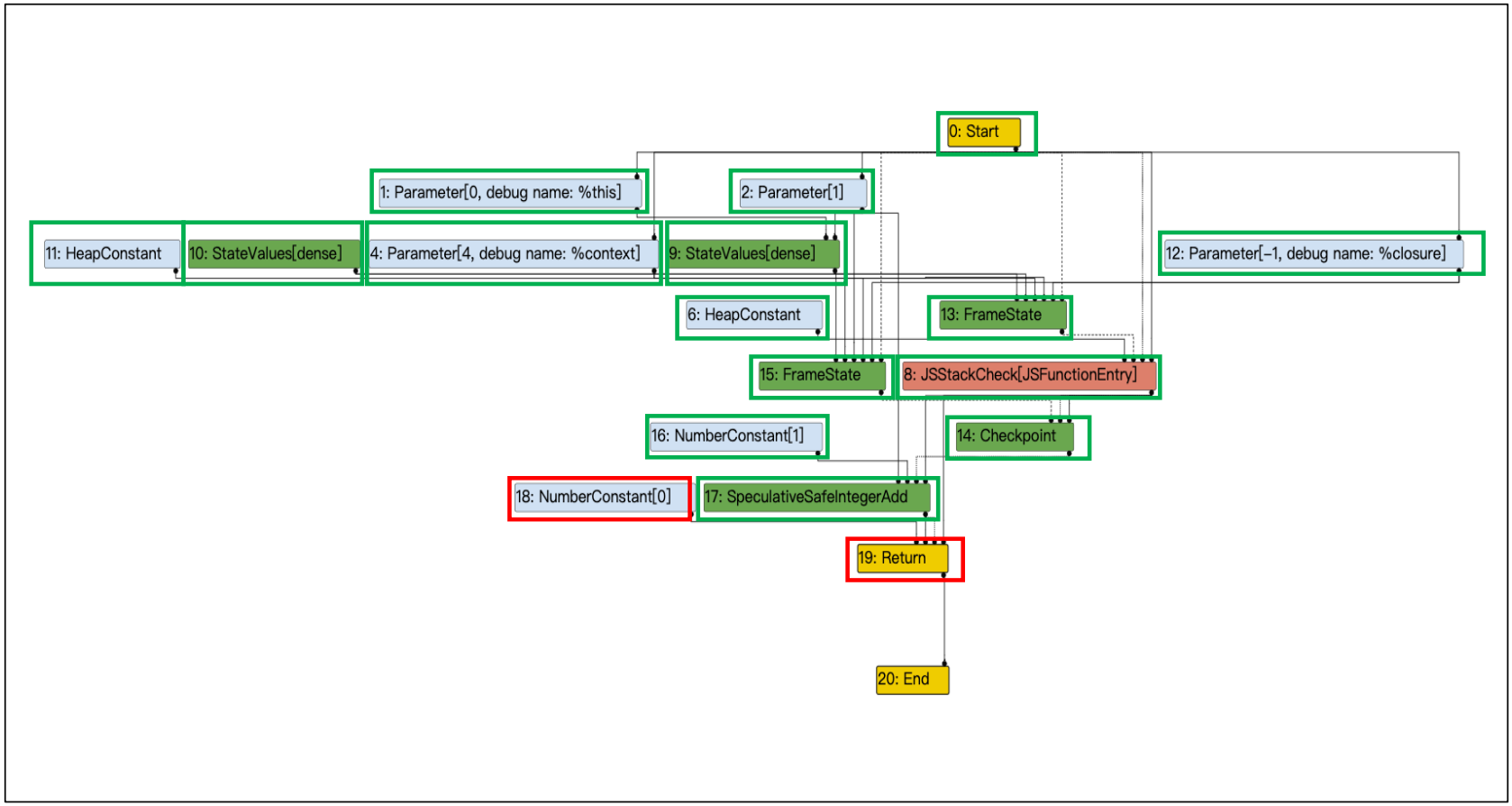
```
JSTypeHintLowering::LoweringResult
BytecodeGraphBuilder::TryBuildSimplifiedBinaryOp(const Operator* op, Node* left,
                                                 Node* right,
                                                 FeedbackSlot slot) {
  Node* effect = environment()->GetEffectDependency();
  Node* control = environment()->GetControlDependency();
  JSTypeHintLowering::LoweringResult result =
      type_hint_lowering().ReduceBinaryOperation(op, left, right, effect, control, slot); //Step7.14
ReduceBinaryOperation
  ApplyEarlyReduction(result); //Step 7.17 更新environment的control和effect依赖状态
  return result;
}
```

```
void BytecodeGraphBuilder::ApplyEarlyReduction(JSTypeHintLowering::LoweringResult reduction) {
  if (reduction.IsExit()) { MergeControlToLeaveFunction(reduction.control()); }
  else if (reduction.IsSideEffectFree()) {
    environment()->UpdateEffectDependency(reduction.effect()); //更新env的effect
    environment()->UpdateControlDependency(reduction.control()); //更新env的control
  } else {
    DCHECK(!reduction.Changed());
  }
}
```

Step7.18: VisitReturn： Return（第三条字节码）

```
void BytecodeGraphBuilder::VisitReturn() {
  BuildReturn(bytecode_analysis().GetInLivenessFor(bytecode_iterator().current_offset()));
}

void BytecodeGraphBuilder::BuildReturn(const BytecodeLivenessState* liveness) {
  BuildLoopExitsForFunctionExit(liveness);
  // Note: Negated offset since a return acts like a backwards jump, and should
  // decrement the budget.
  BuildUpdateInterruptBudget(-bytecode_iterator().current_offset()); //还记得在解释器中介绍过的用于TierUp进入时机判断的InterruptBudget么？它被初始化成一个大于0的量，在字节码执行过程中，遇到向后跳转或者顺序执行就减少，遇到向前跳转就增加，减成0之后就开始判断是否可以TierUP JIT。如果当前的JIT过程嗨可以TierUP，就这里把整个test函数的字节码size从Budget中减去
  Node* pop_node = jsgraph()->ZeroConstant(); //创建返回值0的常量节点
  Node* control =
      NewNode(common()->Return(), pop_node, environment()->LookupAccumulator()); //创建return节点
  MergeControlToLeaveFunction(control); //exit control的merge
}

void BytecodeGraphBuilder::MergeControlToLeaveFunction(Node* exit) {
  exit_controls_.push_back(exit); //将return节点加入SON图的exit_controls_, SON图的exit节点必须受控于该return节点
  set_environment(nullptr); //env置成null
}
```

已构建完成的节点


0: Start
1: Parameter[0, debug name: %this]
2: Parameter[1]
11: HeapConstant
10: StateValues[dense]
4: Parameter[4, debug name: %context]
9: StateValues[dense]
12: Parameter[-1, debug name: %closure]
6: HeapConstant
13: FrameState
15: FrameState
8: JSStackCheck[JSFunctionEntry]
16: NumberConstant[1]
14: Checkpoint
18: NumberConstant[0]
17: SpeculativeSafeIntegerAdd
19: Return
20: End

节点列表

0	Start	
1	Parameter	*this
2	Parameter	1
3	HeapConstant	UndefinedConstant（用于values_向量中的register和accumulator初始值）
4	Parameter	context
5	HeapConstant	指向FeedBackVector
6	HeapConstant	nativecontext
7	Dead	JSStackCheck的一个Dummy输入
8	JSStackCheck	
9	StateValues	StateValues for parameter state, 作为FrameState的第一个输入
10	StateValues	StateValues for register states, 作为FrameState的第二个输入
11	HeapConstant	StateValues for accumulator states,作为FrameState的第三个输入
12	Parameter	StateValues for function closure states,作为FrameState的第四个输入
13	FrameState	替换JSStackCheck的一个Dummy输入
14	CheckPoint	作为AddSmi操作的Eager Checkpoint
15	FrameState	作为14号Checkpoint的FrameState输入
16	NumberConstant	AddSmi的常量操作数1
17	SpeculativeSafeIntegerAdd	加法的节点
18	HeapConstant	返回值0
19	SpeculativeSafeIntegerAdd	加法操作

Step by Step, Node by Node讲述图的构建过程

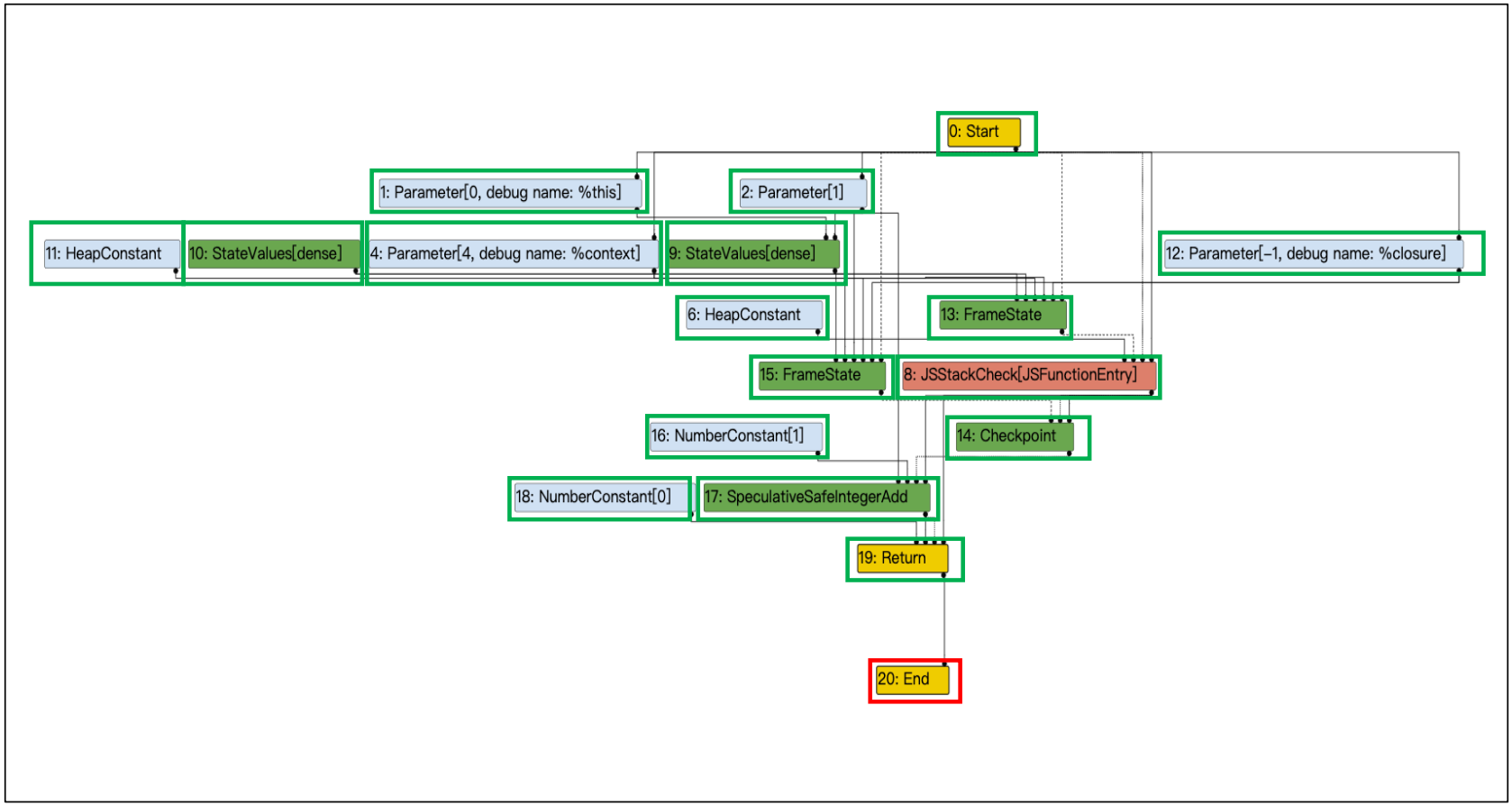
1. new and set the Start node
2. new Environment and set_environment
3. CreateFeedbackCellNode
4. CreateFeedbackVectorNode
5. MaybeBuildTierUpCheck
6. CreateNativeContextNode
7. VisitBytecodes: one by one
8. new and set the End node

Step 8: new and set the End node

@src/compiler/bytecode-graph-builder.cc

```
void BytecodeGraphBuilder::CreateGraph() {
  … …
    // Finish the basic structure of the graph.
  DCHECK_NE(0u, exit_controls_.size());
  int const input_count = static_cast<int>(exit_controls_.size());
  Node** const inputs = &exit_controls_.front();
  Node* end = graph()->NewNode(common()->End(input_count), input_count, inputs); //创建End节点，End节点的输入是该图所有的exit_control节点
  graph()->SetEnd(end); //set End node
}
```

已构建完成的节点


0: Start
1: Parameter[0, debug name: %this]
2: Parameter[1]
11: HeapConstant
10: StateValues[dense]
4: Parameter[4, debug name: %context]
9: StateValues[dense]
12: Parameter[-1, debug name: %closure]
6: HeapConstant
13: FrameState
15: FrameState
8: JSStackCheck[JSFunctionEntry]
16: NumberConstant[1]
14: Checkpoint
18: NumberConstant[0]
17: SpeculativeSafeIntegerAdd
19: Return
20: End

节点列表

0	Start	
1	Parameter	*this
2	Parameter	1
3	HeapConstant	UndefinedConstant（用于values_向量中的register和accumulator初始值）
4	Parameter	context
5	HeapConstant	指向FeedBackVector
6	HeapConstant	nativecontext
7	Dead	JSStackCheck的一个Dummy输入
8	JSStackCheck	
9	StateValues	StateValues for parameter state, 作为FrameState的第一个输入
10	StateValues	StateValues for register states, 作为FrameState的第二个输入
11	HeapConstant	StateValues for accumulator states,作为FrameState的第三个输入
12	Parameter	StateValues for function closure states,作为FrameState的第四个输入
13	FrameState	替换JSStackCheck的一个Dummy输入
14	CheckPoint	作为AddSmi操作的Eager Checkpoint
15	FrameState	作为14号Checkpoint的FrameState输入
16	NumberConstant	AddSmi的常量操作数1
17	SpeculativeSafeIntegerAdd	加法的节点
18	HeapConstant	返回值0
19	SpeculativeSafeIntegerAdd	加法操作
20	End	

从Bytecode到SON图的构建：总结

- Demo case 及其 Graph
- 总体构建流程概述
- Step by Step, Node by Node讲述图的构建过程
  1. new and set the Start node
  2. new Environment and set_environment
  3. CreateFeedbackCellNode
  4. CreateFeedbackVectorNode
  5. MaybeBuildTierUpCheck
  6. CreateNativeContextNode
  7. VisitBytecodes: one by one
  8. new and set the End node

- 了解了一个简单的case的“复杂地”SON图构建流程
- 了解了SON图的结构和构建的相关函数
- 为后续讲解TurboFan的SON图变换和优化打下基础

谢 谢

欢迎交流合作