Graal IR: An Extensible Declarative Intermediate Representation

Gilles Duboscq* Lukas Stadler* Thomas Würthinger†
Doug Simon† Christian Wimmer† Hanspeter Mössenböck*

*Institute for System Software, Johannes Kepler University Linz, Austria †Oracle Labs
duboscq@ssw.jku.at stadler@ssw.jku.at thomas.wuerthinger@oracle.com
doug.simon@oracle.com christian.wimmer@oracle.com moessenboeck@ssw.uni-linz.ac.at

Abstract

We present an *intermediate representation* (IR) for a Java *just in time* (JIT) compiler written in Java. It is a graph-based IR that models both control-flow and data-flow dependencies between nodes. We show the framework in which we developed our IR. Much care has been taken to allow the programmer to focus on compiler optimization rather than IR bookkeeping. Edges between nodes are declared concisely using Java annotations, and common properties and functions on nodes are communicated to the framework by implementing interfaces. Building upon these declarations, the graph framework automatically implements a set of useful primitives that the programmer can use to implement optimizations.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization

General Terms Algorithms, Languages, Performance

Keywords Java, compilation, intermediate representation

1. Introduction

The intermediate representation (IR) is one of the central building blocks of a compiler and significantly impacts the design of the compiler. The way the IR is structured and presented to the programmer influences the way in which the different IR transformation phases of the compiler are written. Some optimization phases can be inherently complex and difficult to implement. The IR framework upon which they are built should be as simple and easy to use as possible so that the focus can stay on the optimizations.

The design of this framework is not only important for the implementation of the compiler, but also for its maintenance. Developers can understand the implementation of a complex optimization faster if the underlying IR framework does not get in their way.

We developed an IR as part of Graal OpenJDK project [11] and its Graal Virtual Machine (VM). The Graal VM is a modification of the Java HotSpot VM which has two JIT compilers: the client compiler [10] and the server compiler [12]. The client compiler aims at fast compilation speed, while the server compiler aims at better optimization at the expense of slower compilation speed. In the Graal VM, a third compiler is added: the Graal compiler, which uses our new IR. It is written in Java and aims to produce highly optimized code. The starting point of the Graal compiler was the design of a new IR for a derivative of the C1X [13] compiler, which uses an IR similar to that of the HotSpot client compiler. The new design simplifies the implementation of standard compiler optimizations as well as aggressive speculative optimizations. An additional goal is to make the IR and the associated infrastructure as easy to use as possible so that compiler developers can focus on implementing new optimizations. We also believe that a clear and easy to use model for the IR can improve the overall maintainability of the compiler.

2. Declarative Intermediate Representation

2.1 Data Flow

The Graal IR is based on a directed graph structure. Each node produces at most one value. It is in static single assignment (SSA) form [6]. Node types are specified in a declarative way through class definitions. All node classes inherit from a base `Node` class. The operation and value represented by a node is defined by its type. The example in Listing 1 defines an `AddNode` type used to represent additions. Edges to other nodes are declared using fields. In this case there are two edges pointing to the operands.

To represent data flow, a node has *input* edges (also called use-def edges) pointing "upwards" to the nodes that produce its operands. Data flow edges are annotated with the `@Input` annotation. The `AddNode` type in Listing 1 has two input edges: `left` and `right`. For nodes with a variable number

Reprinted from APPLC '13, [Unknown Proceedings], February 23, 2013, Shenzhen, China, pp. 1–9.

```
class AddNode extends Node {
  @Input Node left;
  @Input Node right;
}
```

Listing 1. Definition of binary add node class.

of input edges, the collection class `NodeInputList` provides a list of inputs that can grow. For example, the declaration of the `PhiNode` in Listing 2 uses a `NodeInputList` for the different values merged by a SSA phi node. Inputs are ordered in an `NodeInputList`, which is important for a phi node.

```
class PhiNode extends Node {
  @Input NodeInputList values;
  @Input MergeNode merge;
}
```

Listing 2. Definition of a phi node class using a `NodeInputList` for the values it merges.

A graphical representation of some node instances is shown in Figure 1. The black hollow arrow heads represent data-flow edges and point towards the input node.

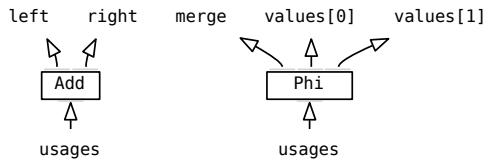

Figure 1. Nodes and data flow edges.

2.2 Control Flow

To represent control flow, a node has *successor* edges pointing "downwards" to its different possible successors. Control flow edges are annotated with the `@Successor` annotation. The `IfNode` type in Listing 3 has two successor edges: `trueSuccessor` and `falseSuccessor`. Like for data flow, a field of type `NodeSuccessorList` is used when the number of successors can vary. For example, the declaration of the `SwitchNode` in Listing 4 uses a `NodeSuccessorList` for the successors corresponding to the different cases.

```
class IfNode extends Node {
  @Input BooleanNode condition;
  @Successor BeginNode trueSuccessor;
  @Successor BeginNode falseSuccessor;
}
```

Listing 3. Definition of an if node class using `@Input` and `@Successor` annotations.

Each block of the control-flow graph begins with a `Begin` node. Two or more lines of control flow can merge at a `Merge` node, which is a subclass of the `Begin` node. These

```
class SwitchNode extends Node {
  @Input Node test;
  @Successor NodeSuccessorList cases;
}
```

Listing 4. Definition of a switch node class using a `NodeSuccessorList` for its successors.

`Merge` nodes need to know the order of their control flow predecessors, so that an SSA phi node can select the correct input value for each of the merge predecessors. As predecessor edges are not ordered, `Merge` nodes are connected to their predecessors using input edges pointing to `End` nodes. These `End` nodes are at the end of the control flow of the merge's predecessors. The `Phi` nodes are attached to their `Merge` node through a special input edge. This structure is illustrated in Figure 2. The red filled arrows represent control-flow edges and point towards the successor nodes.

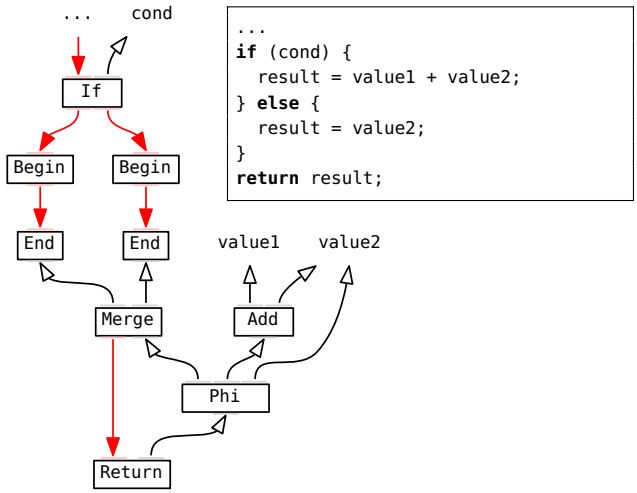

Figure 2. Example graph with control-flow and data-flow edges.

For simplicity, the IR only represents reducible loops. Common Java compilers that translate source code to bytecode never create irreducible loops. The IR models loops explicitly: the loop header is a `LoopBegin` node. This is the only entry point into the loop body. The back-edges of a loop are represented with `LoopEnd` nodes, which are attached to their `LoopBegin` through an input edge. Since the `LoopBegin` node merges the control flow of the loop pre-header and back-edges, `Phi` nodes can be attached to `LoopBegin` nodes. This structure is illustrated in Figure 3 with a simple `for` loop.

Since `LoopBegin` nodes merge control flow, `LoopBegin` node is a subclass of `Merge` node and `LoopEnd` node is a subclass of `End` node.

As seen with these nodes, extensibility is achieved using inheritance: `LoopBegin` nodes build upon the features of

`Merge` nodes. This is particularly useful in this case because it simplifies the implementation of phi nodes. For example, in Listing 5 we can see that the interface used to relate phi nodes and merge nodes works in the same way whether it is a phi of a loop or of a normal merge.

```
class PhiNode extends Node {
  ...
  public MergeNode merge()
  public ValueNode valueAt(EndNode pred)
}
```

Listing 5. Excerpt of the interface used for phi nodes. These methods are used to access the fields shown in Listing 2.

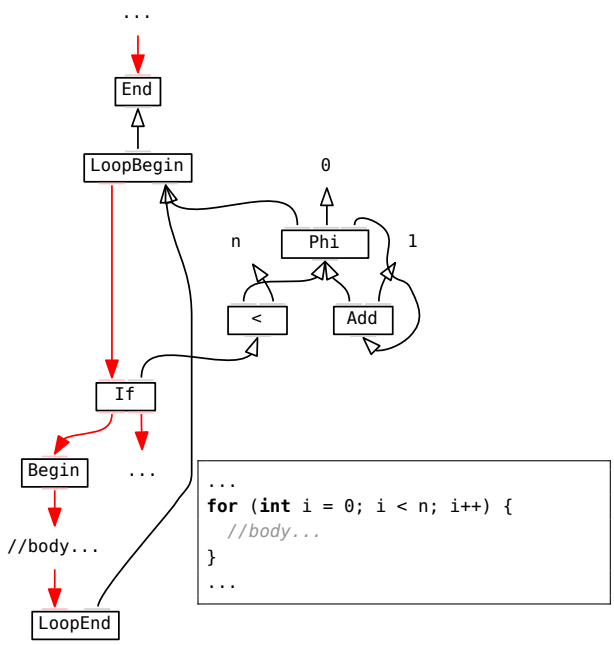

Figure 3. Example graph for a loop.

Note that successor edges are never used for anything that is not strictly a direct control flow successor. Input edges, however, are sometimes used for purposes other than pure data flow. All input edges express a scheduling dependency. That is, a node must be scheduled after all its dependencies when emitting code.

2.3 Fixed and Floating Nodes

Nodes are not necessarily fixed to a specific point in the control flow. The control-flow splits and merges provide a backbone around which most other nodes are *floating*. For example, the `Add` node in Figure 2 is floating. These floating nodes are only constrained by their data-flow edges and through additional dependencies such as memory dependencies. The constraints maintain the program semantics but allow more freedom of code motion for operations. When a node needs to express a dependency to a specific branch, it has an input edge pointing to a specific `Begin` node.

While building the graph, we create floating nodes for all nodes where this does not require any special analysis or dependency creation. Some operations that cannot immediately be represented as floating nodes, such as memory read operations, are transformed into floating nodes by later optimization phases that insert the appropriate dependencies.

As explained by Click [3], this representation simplifies a number of optimizations by removing the burden of maintaining a valid schedule or performing code motion. Before emitting machine code, the IR is fully scheduled by assigning each node to a block in the control-flow graph and by ordering the nodes inside each block. During scheduling, simple heuristics are used to apply some code motion optimization such as hoisting code out of loops.

In order to include the scheduling information in the graph, all IR nodes that can be scheduled inherit from `ScheduledNode`, which has a `next` successor edge. The scheduler just sets this pointer. If we want to throw away the schedule, we can just null out this pointer for all floating nodes. This can be useful to temporarily schedule the graph for a specific optimization.

3. Graph Infrastructure

Our declarative framework allows the graph infrastructure to know about the different node types and their edges. The graph infrastructure then uses this knowledge to provide extra functionality that is described below. As annotations and the usage of fields for edges allow a concise node declaration, the extra functionality enabled by the graph infrastructure allows the whole compiler to be concise. This section presents the concepts that our infrastructure provides, while Section 4 shows how these concepts are implemented efficiently.

3.1 Reverse Edges

For both edge types, the reverse edges are automatically maintained. The reverse edges for input edges are called *usage* edges (or def-use edges). The reverse edges for successor edges are called *predecessor* edges. In the Graal IR, as a result of the control flow model, there can only be at most one predecessor[1].

Reverse edges are used by compiler optimizations. In addition, they are used to navigate between related elements of the IR. For example, the `LoopEnd` nodes associated with a `LoopBegin` node can be found in this `LoopBegin` node's usages.

The reverse edges are automatically updated when a new node is inserted in the graph. However, if we make an explicit change to a node's edges, the graph must be notified to properly update the reverse edges. Using setter methods for

[1] Recall that merge nodes refers to the merged branches through inputs, not predecessors.

these fields mitigates the problem (see Listing 6). There are notifications for input edges (`updateUsages`) as well as for successor edges (`updatePredecessor`). Note that the graph is automatically notified when using a `NodeInputList` or a `NodeSuccessorList`.

```
class Node {
  protected void updateUsages(Node oldInput, Node newInput)
  ...
}

class IfNode extends Node {
  @Input BooleanNode condition;
  ...
  public void setCondition(BooleanNode x) {
    updateUsages(condition, x);
    condition = x;
  }
  ...
}
```

Listing 6. Setter method for an edge of the if node. This setter notifies the graph of an input change through the `updateUsages` method.

While the direct edges are named and can be distinguished from one another, reverse edges are not. For example, in Figure 2, we know that the `Add` node is used by the `Phi` node but we do not know if this usage comes from the `merge` field or the `values` field of the `Phi` node. Even if one could deduce that it can not be the `merge` field because of the field's type, it could be any index in the `values` input list. To discover the precise input from which this usage comes, one would have to iterate over the `Phi` node's inputs.

Also, note that the usages of a node are not maintained as a set but rather as a multiset. If a node A is used by more than one input edge of node B, then B appears more than once in the usages of A.

3.2 Edge Iterators

All the different edge types can be iterated as seen in the API excerpt of Listing 7. The direct edges can be iterated and support modification to existing edges during iteration. If an edge is modified during iteration before it has been processed, the updated value is processed. Adding or removing an edge from a `NodeInputList` or a `NodeSuccessorList` during iteration is not supported. The nodes from a `NodeInputList` or a `NodeSuccessorList` are seen in their original order during iteration.

The iterator for the direct edges iterates over the values of the edges (that is the nodes referenced by the edges) but it can also be used to iterate over `Position` objects. A `Position` object represent the edge itself. It can be used to get the value of the same edge on another node or to get the name of the edge (that is the name of the field). This is supported by the `NodeClassIterator` returned by `NodeClassIterable` (see Listing 7).

```
class Node {
  public NodeClassIterable inputs()
  public NodeClassIterable successors()
  public Node predecessor()
  public NodeIterable<Node> usages()
  ...
}

abstract class NodeClassIterable extends
     AbstractNodeIterable<Node> {
  @Override
  public abstract NodeClassIterator iterator();
}

class NodeClassIterator implements Iterator<Node> {
  public Position nextPosition()
  ...
}
```

Listing 7. Edge iterators. (`NodeIterable` is explained in Section 3.6 and Listing 11)

For reverse edges, since there can be only one predecessor, this predecessor can be accessed directly. The usages can be iterated but their order is not specified. Modifying the usages during iteration of reverse edges is not supported.

3.3 Node Replacement

The explicit and declarative specification of edges allows methods that change the first edge that points to a specific node to point to another node. For this purpose we have two methods in the `Node` class: `replaceFirstInput` and `replaceFirstSuccessor` (see Listing 8).

```
class Node {
  public void replaceFirstInput(Node oldInput, Node
       newInput)
  public void replaceFirstSuccessor(Node oldSuccessor, Node
       newSuccessor)

  public void replaceAtUsages(Node other)
  public void replaceAtPredecessor(Node other)

  public void replaceAndDelete(Node other)
  ...
}
```

Listing 8. Node Replacement.

Another useful replacement is to change all edges pointing to a specific node to point to a different node. This is done through `replaceAtUsages` and `replaceAtPredecessor`. Finally we can completely replace a node by another one by using the `replaceAndDelete` method. All these methods automatically update the reverse edges to account for the changed direct edges.

Replacement is useful for example when replacing high level nodes by lower level nodes. In the Graal compiler this is called *lowering*. The lowering process allows the IR to have different granularities during compilation. For exam-

ple, during lowering, a `LoadField` node is transformed into a null-check guard and a memory `Read` node. This is easily implemented by *replacing* the `LoadField` node by a new `Read` node. Replacement is also useful for implementing constant folding where the folded operation is *replaced* by the resulting constant.

3.4 Node Cloning

The knowledge about edges also allows the graph infrastructure to implement node, graph, or sub-graph cloning easily. One of the interesting applications of cloning is method inlining where the graph of the inlined method is cloned into the graph of the caller. Cloning is also used by transformations that duplicate code such as tail duplication as well as many loop transformations.

```
class Node {
  public final Node copyWithInputs()
  ...
}

class Graph {
  public Map<Node, Node> addDuplicates(Iterable<Node>
       nodes, Map<Node, Node> replacementsMap)
  ...
}
```

Listing 9. Node Cloning.

The `copyWithInputs` method clones into the same graph and preserves the input edges. This is useful when cloning just one node. In order to clone more than one node at a time, the `addDuplicates` method can be used. This method can clone nodes from a different graph and returns a mapping from the original nodes to the new duplicates.

The edges between the original nodes are preserved in the cloned nodes. If an edge of an original node points to a node that is not part of the duplicated ones, it is cleared in the cloned node. The `replacementsMap` parameter allows to replace some of the original nodes with nodes that are already in the target graph. For example, during inlining this is used to replace the formal parameters of the inlined method's graph by the actual parameters from the calling method's graph.

3.5 Typed Iterators

Many optimization phases benefit from quickly accessing certain types of nodes in the graph. For example, a phase that transforms loops is interested in iterating over all `LoopBegin` nodes in the graph. This becomes more important with larger graphs. The fast iteration over all nodes of a certain type is available for all node classes that implement the interface `IterableNodeType` (see for example `LoopBegin` in Listing 10). Then it becomes possible to use the `getNodes` method of the `Graph` class described in Listing 10 to iterate over these nodes.

This method has linear complexity with the number of nodes of the requested type in the graph, this is, it does not iterate all nodes of the graph or perform a type check on every node. If there are n nodes of type `T` in a graph then `getNodes(T.class)` has a $\mathrm{O}(n)$ complexity. Note that the `IterableNodeType` interface is just a marker interface and does not contain any methods.

```
class Graph {
  public <T extends Node & IterableNodeType>
       NodeIterable<T> getNodes(final Class<T> type)
  ...
}

class LoopBeginNode extends MergeNode implements
     IterableNodeType {
  ...
}

for (LoopBeginNode loop :
     graph.getNodes(LoopBeginNode.class)) {
  ...
}
```

Listing 10. Typed Iterator.

During iteration it is possible to delete a node of the iterated type from the graph. If this happens, this node is not processed by the iterator. Also, a node of the iterated type added to the graph while iterating it is processed by the current iterator. This behavior is useful when the compiler needs to process all nodes of a certain type while inserting new nodes of this same type during the transformation.

Typed iterators are polymorphic: they return all nodes of the requested type and nodes of subclass of the requested type.

3.6 Node Iterators

Most of the iterable collections implement the `NodeIterable<T>` interface. This interface adds a number of helper methods (see Listing 11).

```
interface NodeIterable<T extends Node> extends Iterable<T> {
  <F extends T> NodeIterable<F> filter(Class<F> clazz);
  List<T> snapshot();
  T first();
  int count();
  boolean isEmpty();
  boolean isNotEmpty();
  boolean contains(T node);
}
```

Listing 11. Iterable Node collection interface.

The ability to filter on a particular type of node is useful to navigate in the IR. For example, the back-edges of loops are modeled by `LoopEnd` nodes that are linked to their `LoopBegin` node using a special input edge. To find all the back-edges of a `LoopBegin` node, we can look for all of its usages that are `LoopEnd` nodes (see Listing 12).

```
class LoopBeginNode extends MergeNode {
  public NodeIterable<LoopEndNode> loopEnds() {
    return usages().filter(LoopEndNode.class);
  }
  ...
}
```

Listing 12. Using a filter operation on a `LoopBegin` node to find its associated `LoopEnd` nodes.

The `NodeIterable` interface also allows to take snapshots of the collection. This is useful to avoid problems if the collection is modified during iteration. Other helper methods such as `first`, `count`, `isNotEmpty`, `isEmpty` and `contains` allow the code using those iterable collections to be concise and descriptive.

3.7 Global Value Numbering

If nodes of a specific type can be replaced with a congruent node (as defined by Alpern et al. [2]), it can be marked with the `ValueNumberable` interface. Nodes of this type can then be used with the graph API shown in Listing 13. The `unique` method can be used when inserting a new node into the graph instead of the standard `add`. If a congruent node is already present in the graph, it is returned, otherwise the new node is added to the graph and returned. Once a node is already in the graph, the `findDuplicate` can be used to find congruent nodes.

```
class Graph {
  public <T extends Node> T add(T node)
  public <T extends Node & ValueNumberable> T unique(T node)
  public Node findDuplicate(Node node)
  ...
}
```

Listing 13. Finding congruent nodes in a graph when inserting a node or later.

The `ValueNumberable` interface is just a marker interface and does not require implementing any methods. Once a node type is marked with it, the graph infrastructure can automatically uses its knowledge of edges when checking for congruency. For node types, the graph infrastructure also knows about fields in nodes that are neither inputs nor successors. These fields are called *properties* and are also taken into account while checking for congruency. In the Graal IR, almost all floating nodes are `ValueNumberable` nodes.

3.8 Serialization

Thanks to the graph infrastructure, we can easily implement graph serialization. The serialization is done by iterating over edges and properties. New node types do not need to do anything to be included in the serialization.

Leveraging this property, we output the graph in different formats such as XML, plain text, and binary. To view the graphs we use the visualizers originally developed for the Java HotSpot server compiler IR [14] and the Java HotSpot client compiler IR [9].

3.9 Extensibility

The declarative style used for node definitions facilitates extensibility by reducing the amount of code needed to add a new node type. When introducing a new node type, all of the benefits of the graph infrastructure only require adding edge annotation, marker interfaces (if needed) and notification (if the node has mutable edges).

To improve extensibility, it is also important to avoid patterns such as the Visitor pattern. To include new node types in the existing optimization phases, a number of interfaces are available. For example:

- `Canonicalizable` can be implemented if the node can take a canonical form depending on the values of its edges. The changes are limited to the replacement or deletion of this node. For example, constant folding is implemented through this mechanism.
- `Simplifiable` can be implemented if the node can be simplified through a large CFG change depending on the values of its edges. For example, this is used to remove an `If` node if the condition is a known constant.
- `Lowerable` can be implemented for a high level node that should be replaced by lower level nodes.
- `LIRLowerable` can be implemented for nodes that need to be transformed into low-level IR for register allocation and code generation.

In these interfaces, a "Tool" is usually provided as a callback into the specific optimization that the node can use. For example, Listing 14 shows the `Simplifiable` interface and the associated `SimplifierTool`. This tool lets the simplified node remove a whole branch from the graph and is provided by the transformation that is calling the `simplify` method.

```
interface Simplifiable {
  void simplify(SimplifierTool tool);
}

interface SimplifierTool {
  void deleteBranch(FixedNode branch);
}
```

Listing 14. The `Simplifiable` interface and its associated `SimplifierTool`.

4. Implementation

4.1 Edges and Edge Iterators

Using fields for the edges integrates well with developer tools and operations such as refactoring. It also provides a compact representation for the nodes. No additional data structure such as an array is needed to store the direct edges.

In Figure 4, we can see the structure of a node. The example is based on an `Add` node as defined in Listing 15. The references to the different edges are stored in the fields of the node object.

```
abstract class ScheduledNode extends Node {
  @Successor private ScheduledNode next;
}

class AddNode extends ScheduledNode {
  @Input private Node left;
  @Input private Node right;
}
```

Listing 15. Definition of an `Add` node that extends `ScheduledNode`.

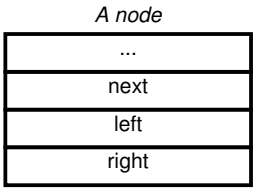

Figure 4. Structure of a node: edges as fields.

In addition to the edge fields, all nodes have a number of other fields used by the graph infrastructure. This is illustrated in Figure 5. The graph infrastructure is implemented using a metaclass system. For each node type, we automatically build a metaclass that describes it. This metaclass contains information about all the edge and property fields. Each node has a reference to its metaclass.

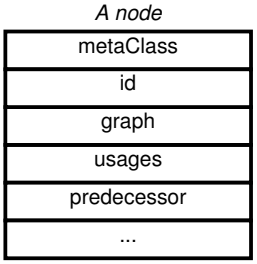

Figure 5. Structure of a node: header for the graph infrastructure.

Nodes have a numeric identifier that is unique in the scope of their graph. This "id" is assigned when the node is added to the graph. Nodes also have a reference to their graph. The graph itself contains an array of its nodes.

Each node has a reference to its list of usages that is maintained when new nodes are added to the graph or when a node changes one of its edges. The other type of reverse edge is the predecessor. Since nodes can have at most one predecessor, they have a direct reference to this predecessor that is maintained in a similar fashion to the usages.

To achieve fast iteration over the edges of a node, the metaclass has a list of offsets where edges can be found in the node type it describes. For example in Figure 6, we can see that the metaclass knows that inputs can be found at offsets 56 and 64 while a successor edge can be found at offset 48. In order to iterate over edges, we just iterate over these arrays of indexes and directly access the edges in the node object using `Unsafe`[2] access.

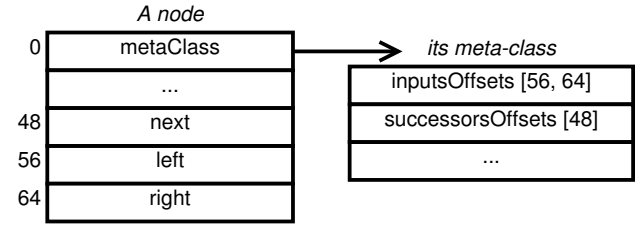

Figure 6. Structure of a node: the metaclass.

4.2 Typed Iterators

In order to implement the fast typed iterators, we use lists of nodes for each node type that implements `IterableNodeType`. This is a simply linked list of node objects. The pointer to the next element is directly embedded in the node class (see Figure 7). This avoid the need for wrapper objects for these lists. There is one list per `IterableNodeType`, which means that all nodes in such a list have the same class and metaclass.

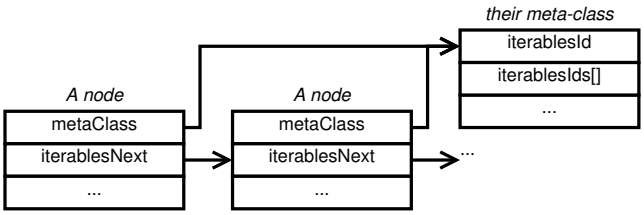

Figure 7. Linked list of nodes implementing `IterableNodeType`.

To find the head and tail of these simple linked lists, each graph has a table for both heads and tails (see Figure 8). To allow fast access, these tables are arrays indexed by a numeric identifier that is unique to each metaclass. This `iterablesId` can be seen in the metaclass in Figure 7. The head is used to start iteration while the tail is used to add new nodes to the list when a node implementing `IterableNodeType` is added to the graph.

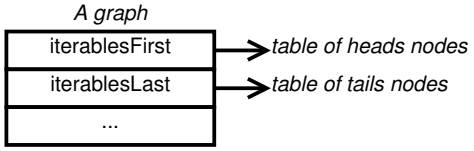

Figure 8. Graph structure containing the head and tail tables.

[2] The `sun.misc.Unsafe` API allows low-level access to Java objects

To support polymorphism for typed iterators, each node metaclass has a list of `iterablesIds` that it should process. This can be seen in the `iterablesIds` array in the metaclass in Figure 7. The typed iterators process all the nodes of each class found in this array. When it reaches the end of the `iterablesIds` array, it goes back at the beginning. For each class, it restarts in linked list from the last node it saw in the last iteration. This process in repeated until no new nodes are found.

This is important because we want the typed iterator to process new nodes that are inserted during iteration. For example, `Merge` is an `IterableNodeType` and `LoopBegin` is a subclass of `Merge`. While iterating over all `Merge` nodes, we start iterating using the linked list of `Merge` nodes, then we use the linked list of `LoopBegin` nodes. If while using the linked list of `LoopBegin` nodes, a new `Merge` node is added, we need to go back to the `Merge` nodes linked list to see it.

Deleting nodes from these lists is done lazily: when a node is deleted from the graph, it is just marked as deleted. Later when some code requires iteration over these lists using a typed iterator, deleted nodes are unlinked while iterating.

5. Results

The source code of the Graal IR and Graal compiler is available as part of the Graal OpenJDK project [11].

To evaluate the granularity of the IR, we measured the number of nodes in IR graphs at different points during compilation in the Graal compiler. We compared those numbers to similar measurements in the HotSpot client and server compilers from HotSpot 24.0-b20 [1]. This evaluation was done using an x86_64 version of the three compilers by running all DaCapo benchmarks [7] (version 9.12) multiple times.

Both the Graal compiler and the server compiler use special nodes to keep deoptimization states (`FrameState` nodes in Graal IR) whereas the client compiler keeps those state as a special data structure. To make a meaningful comparison, these client compiler data structures are counted as IR nodes. The results are shown in Figure 9.

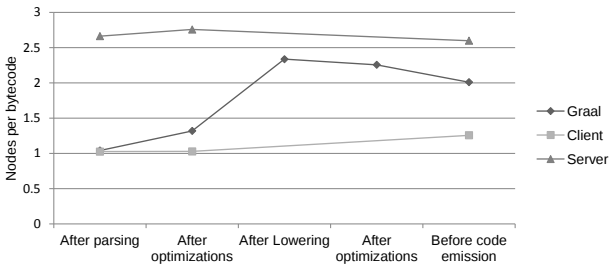

Figure 9. Number of nodes per bytecode at various steps of the compilation for different compilers.

After parsing the methods, the size of the Graal IR is similar to the size of graphs in the client compiler. This can be explained because both have similar high-level IRs at this point. The server compiler IR, on the other hand, is already at a lower level and has significantly more nodes per bytecode.

In both the Graal compiler and the server compiler, the IR grows during optimization because of transformations that duplicate code (loop optimizations and tail duplication). The client compiler does not perform such transformations and thus the size of its IR remains almost the same after optimization.

As expected, the lowering phase expands the number of nodes in the Graal IR, bringing it closer to the size of the server compiler IR. Further optimizations in the Graal compiler slightly lower the number of nodes again due to simplifications.

When bringing the IR close to machine code ("Before code emission"), the high-level nodes of the client compiler IR are split into lower-level operations, increasing the size of the IR. For the Graal compiler and the server compiler, the already low-level IRs shrink during this operation. In the server compiler this is done using a bottom-up rewrite system, which selects platform-specific instructions [12]. On the x86 platform, the use of complex instructions can explain this shrinking. In the Graal compiler the IR shrinks because nodes that do not emit code (such as for example `FrameState` nodes, `Begin` nodes) are discarded.

6. Future Work

To improve the precision of node declarations, we plan to support for commutative edges. This would allow the graph to consider two addition operations congruent even if their inputs are swapped.

7. Related Work

The Graal IR is related to the graph IR presented by Click [4, 5] where nodes are not necessarily fixed to a specific point in the control flow. This kind of IR retains ideas from the Program Dependence Graph (PDG) of Ferrante et al. [8] where only the dependencies necessary to express the program semantics are kept.

A notable difference to Click's model is the different direction of edges for control flow and data flow. This property allows our IR to avoid projection nodes in the control flow graph. In Click's IR, such nodes are necessary in order to differentiate the successors of a control split. In our IR, since a control split points to its successors, projection nodes are not needed.

Also, in Click's IR, edges of a node are referenced with integer indexes. We use named edges in order to improve the maintainability of the IR itself as well as the maintainability of code that uses the IR.

8. Summary

We have seen how the declarative style of an IR can convey useful information to an underlying framework while mak-

ing the declaration of IR nodes clear and concise. We have also seen how using this information allows the IR framework to provide useful functions to the compiler.

This declarative style and this framework help keeping declaration of new nodes easy and also serves as an interface to develop the transformation phases of the compiler. As part of the IR framework we have also implemented an efficient way of iterating over all nodes of a specific type.

Acknowledgments

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

References

[1] HotSpot Express 24, build 20. URL `http://hg.openjdk.java.net/hsx/hsx24/hotspot/rev/6d0436885201`.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 1–11. ACM Press, 1988. ISBN 0-89791-252-7. doi: 10.1145/73560.73561.

[3] C. Click. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–257. ACM Press, 1995. ISBN 0-89791-697-2. doi: 10.1145/207110.207154.

[4] C. Click and M. Paleczny. A simple graph-based intermediate representation. In *Papers from the ACM SIGPLAN workshop on Intermediate representations*, IR '95, pages 35–49. ACM Press, 1995. ISBN 0-89791-754-5. doi: 10.1145/202529.202534.

[5] C. N. Click, Jr. *Combining Analyses, Combining Optimizations*. PhD thesis, Rice University, 1995. URL `http://hdl.handle.net/1911/16807`.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991. ISSN 0164-0925. doi: 10.1145/115372.115320.

[7] DaCapo Project. *The DaCapo Benchmark Suite*, 2012. URL `http://dacapobench.org/`.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987. ISSN 0164-0925. doi: 10.1145/24039.24041.

[9] Java.net. Java HotSpot client compiler visualizer, 2012. `http://java.net/projects/c1visualizer/`.

[10] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1):7:1–7:32, May 2008. ISSN 1544-3566. doi: 10.1145/1369396.1370017.

[11] OpenJDK Community. *Graal Project*, 2012. URL `http://openjdk.java.net/projects/graal/`.

[12] M. Paleczny, C. Vick, and C. Click. The Java HotSpot™ server compiler. In *Proceedings of the Symposium on Java Virtual Machine Research and Technology*, pages 1–12. USENIX, 2001.

[13] B. L. Titzer, T. Würthinger, D. Simon, and M. Cintra. Improving compiler-runtime separation with XIR. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 39–50. ACM Press, 2010. ISBN 978-1-60558-910-7. doi: 10.1145/1735997.1736005.

[14] T. Würthinger, C. Wimmer, and H. Mössenböck. Visualization of program dependence graphs. In *Proceedings of the International Conference on Compiler Construction*, pages 193–196. Springer-Verlag, 2008. ISBN 3-540-78790-9, 978-3-540-78790-7.