Thinking About Intermediate Representations

John Rose, September 2014

Optimizing compilation requires subtle algorithms and data structures. Foremost of these is the choice of one or more **intermediate representations** (IR). There is a wide variety of choices available, as exemplified by JVMs (HotSpot C1 and C2, JRockit, J9), Java-on-Java compilers (Graal, Jikes), and static compilers (gcc, llvm). In this document, we compare those choices by describing common IR features. We also narrow down the choices by enumerating design requirements for a idealistic IR we would enjoy working with. We will list the requirements first, because they are the most interesting results. The rest of the document will build up terminology and rationale for these requirements.

Requirements

Transformable

The IR must support transformations flexibily and efficiently, on single nodes, subgraphs, and wholesale. Implementation code for transformations must be clear and concise. (It should also be unit-testable.) Put another way, the design and implementation of the IR must help programmers detect when program semantics might not be conserved under a proposed transformation. If transformations mutate the IR, the mutations must provably exclude side-effects that could change semantics.

Local

The meaning of an IR node `B` must depend (as much as possible) on the meaning of nodes directly connected to it by edges `A -> B` and `B -> A`. If a node depends for its meaning somehow on non-immediate neighbors, that coupling must either be fully present in the immediate neighbors, or carefully documented and respected by all transformations.

Unified

A suite of IRs used for optimizing compilation represents a wide range of constructs, from methods to bytecodes, down to simple operations and machine primitives. The implementation code, concepts, and terms for the IRs in a JVM toolchain should be as uniform as possible, to allow engineers to learn each IR (or each level of IR) more quickly by relying on common notions and notations. (This requirement does not force or rule out a single grand unified IR.)

Multi-level

Some optimizations can be adequately on at or near the granularity of source code or bytecode. IR should not be prematurely lowered to idealized RISC-level or machine level until after coarse grained or source-level optimizations are finished. (This requirement requires that an IR suite be multi-level, not necessarily a single grand unified IR.)

Explicit

Causal dependencies in the IR must be represented and defended explicitly. Specifically, if an IR node `A` performs a side effect, it must be easy to locate the frontier of nodes `B` which are causally effected by the first node `A`. (This allows transforms which replace `A` to find and edit all affected nodes.)

Tractable

The IR must allow optimization logic to require space and time within engineered limits. Theoretical complexity of algorithms should be limited to quasi-linear in size of whole input code. Basic node-level IR operations must be quasi-linear in node complexity. Graph-level IR operations (e.g., compute and cache derived dependencies or types) must be quasi-linear in graph complexity. (Quasi-linear means we can expect `O(N × polylog N)` for input size `N` almost always. More complex algorithms, e.g., quadratic or brute force, may be squeezed in but only for limited scales.)

Efficient

The IR must be implemented with a reasonably small computation cost per size unit of input code, for expected optimization loads, including operations of creation, transformation, and traversal. The IR implementation should avoid excessively diffuse or pointer-rich data structures, excessive population or allocation rate of Java objects, and excessive method polymorphism. Compilation loads can always expand (via inlining or frequent recompilation) to absorb any resource budget, so more efficient optimization directly enables more pervasive optimization.

Java-expressible

The IR must have an efficient implementation in Java. The Java API must be well documented and natural to use and extend in normal-looking Java code. (Note: This requirement could in principle be relocated to a different language such as C++, but is a natural one for working in the context of the JVM.)

Inspectable

The IR must support a human-readable representation that is readily available at any point.

Serializable

The IR must support a serializable representation, for unit testing and (perhaps) live replay. Read and write methods must be high-fidelity. Binary and textual forms are desirable.

Definitions

- Starting from the basics, we observe that a computer runs code to process data.
- **Code** defines the steps taken by the computer and the data it processes.
- Code also includes **metadata** which associates names and other metadata with the basic program.
- **Source code** consists of symbolic text, that can be read and written by humans.
- **Native code** consists of binary instructions that can be efficiently executed by hardware, with associated metadata.

- Java **class files** include data definitions (classes), bytecodes, and other metadata.

Compilation pipeline

- There is a step-by-step process, the **compilation pipeline**, that transforms source code to native code. (The source code is **compiled** and the native code is **generated**.)
- Some of the pipeline works **off-line** long before the code is run, and some on-line. Class files are usually generated off-line by **javac**. Native code is usually generated on-line by a **JIT**.
- Each pipeline phase (except perhaps the first) works on a characteristic **IR**, or "intermediate representation" of the code.
- An IR used for storage (such as class files) has a **serialized** presentation.
- IRs are typically processed by means of in-memory data structures, whether serialized or not.
- Pipeline inputs and outputs should be "tappable", for inspection, recording, and other measurement. (The IR should therefore support high-fidelity serialization and human-readable presentations.)
- Pipeline phases must allow flexible composition.
    - Phases must be individually correct, preserving semantics from input to output. (This property must be in some way provable, at least by code inspection and unit tests.)
    - Some phases run only once (parsing, lowering).
    - Some phases run iteratively until the code "settles" (incremental inlining, loop transformation).

IR graph nodes

- Every IR is structured as a graph of **nodes**. (IRs which are primarily serialized stretch this view, but it can accommodate them.)
- A node represents a separable and distinct part of the code, with its associated behavior.
- Example node meanings:
    - in general: data processing, memory access, control flow (including loops)
    - **source node**: source code operator, constant, name, declaration, statement, method, class.
    - **machine node**: native (hardware) instruction, instruction template (macro-like instruction group), constant, call site, block or method boundary
    - **bytecode node**: similar to native code, but for JVM bytecodes instead of hardware (includes exceptional control and managed pointers)
    - during optimization: cross-platform **ideal node** (RISC-type operation), control-dependent **constraint node** (records inferences), **call node** (inlined or not, constant or not)
    - effects containment: **memory node** (memory effects merge), SSA support such as **phi node** (definition merge) or **lambda node** (use split), **box node**

IR graph edges

- An **edge** between nodes `A` and `B` represents some sort of direct causal relation or **dependency** between `A` and `B`.
- Using spatial metaphors, we say that causes are **upward** or **early** in the graph, while their effects are **downward** or **later** in the graph. The eventual execution of the machine instructions generated from the graph will flow downward, relative to the graph.
- Example edge meanings:
    - **Control dependency**: Node `B` must follow node `A`, in normal execution order.
    - **Exception edge**: Node `B` follows node `A` if `A` throws an exception.
    - **Data dependency**: Node `B` requires a value from node `A`.
    - **Memory dependency**: Node `B` requires a memory state from node `A`.
    - **Dominator**: Node `A` is the lowest node that must execute before `B`.
    - **Post-dominator**: Node `B` is the highest node that must execute after `A`, in normal execution order.
    - **Loop member**: Node `B` is a member of the loop represented by `A`. (`A` dominates `B` and they are connected "strongly" by a cycle, and any cycle through `B` also includes `A`.)
    - **Anti-dependence**: Node `A` must a read memory state before `B` posts side effects to an overlapping state.
- Edges are oriented but logically bidirectional. It is conventional to write `A -E-> B` if nodes `A` and `B` are related by an edge of type `E`, where `A` is above `B` in the graph.
- Implementations always have a preferred traversal direction (upward or downward) for any given edge type, but must also provide ways to traverse a node's edges in the non-preferred direction. (I.e., the IR must support both downward and upward traversal, since some optimization tactics work from effect to cause, while others work from cause to effect.)

Node values

- The behavior of a node `A` produces one or more partial results from running the node's code, called the node's **values**.
- A minimal example of a node behavior is a single native instruction. (Before register allocation it might be `addl t1, t2`.) Such a node's value could be the result loaded by the instruction into the instruction's output machine register (`t1`).
- The input values to that instruction would be expressed as **input edges** relating the node to nodes which must be previously executed.
- Later nodes using the instruction's value would be linked to the node as **output edges**.

Transformations

- An **AST** ("abstract syntax tree") is an IR derived from parsing something (whether textual or binary)
- An AST is a tree, whose internal nodes have an in-degree of one, and an intrinsic parse order.

- In principle, native code could be generated by walking an AST and generating native code directly.
- The best native code is obtained by **transforming** a graph between source parsing and native code generation.
- Transformations must conserve the semantics of the program, but there are many, many tactics available.
- Basic transformations include:
    - **Reordering** a node (or subgraph). (It may move up towards a dominating placement or down towards a theoretical post-dominator. Data operands may also be reordered in order to canonicalize expressions.)
    - **Splitting** a node (or subgraph) into two or more nodes (or subgraphs). (Each split performs the job of the original, but for a subset of the graph affected by the original.)
    - **Merging** two or more nodes (or subgraphs) into one. (This is the opposite of splitting.)
    - **Replacing** a node (or subgraph) by an equivalent. (Common cases are canonicalization, strength reduction, lowering, inlining, and loop reorganization.)
- Transformations may be **local**, making changes within a subgraph of small diameter (just a few edge-hops between involved nodes). Such transformations are valuable because they are relatively easy to prove correct.
- Local transformation examples:
    - Constant fold `2+3` to `5`, canonicalize `(X+2)+3` to `X+5`, strength reduce `X*8` to `X<<3`.
    - Replace a named constant with its value.
    - Replace a small “diamond” with a conditional move.
    - Remove a test or merge it into a pre-existing test.
- Or, transformations may be **non-local**, replacing one subgraph (or the whole graph) by another.
- Non-local transformation examples:
    - **Parse** source or bytecode, creating a new AST or graph from nothing.
    - Copy a whole subgraph (for inlining or loop splitting). A variant copy is deserializing a subgraph.
    - **Lower** one type of IR (subgraph or whole program) to another.
    - **Match** idealized IR to available machine node types. (This is typically done on the whole program, leaving none of the original behind.)
- Most valid transformations are unimportant; most important transformations involve hot regions of the program.
- Transformation effort should therefore be focused on regions of the code where improvements will noticeably affect the whole program.
- Transforms which must be performed globally should be either cheap (CCP, GVN) or inescapable (parsing).
- An AST can in principle be transformed to a more optimal AST for better code generation, but this is awkward.
- It is more helpful to work with a more general form of graph, with less intrinsic order.

IR design considerations

- To design an IR, one must define its **nodes** and **edges**, with their various kinds and meanings.
- One must also define **decorations** for the nodes and edges.
- Note that an IR (like source or native code) is not freely composable from its elements; there are structural **constraints**.
- Node behaviors can correspond to source constructs (AST nodes), **idealized** operations (array range check), or native instructions.
- These choices are not mutually exclusive; when they co-exist **lowering** transforms convert from source-oriented constructs to intermediate idealized constructs to native instructions.
- Higher-level nodes are compact and easy to verify, while lower-level nodes permit a wider range of transforms.
- Decorations can include input attributes like historic profiles, source language types, or source debugging information.
- Decorations can also include derived attributes like **IR types**, **register allocations**, and **instruction selections**.
- An IR graph admits **temporarily derived** elements, which are edges, nodes, or decorations that are computed from input nodes and edges. (These derived elements are often stored in side tables for specific global transformations, or computed lazily, or temporarily inserted and later removed.)
- Examples:
    - A node's immediate dominator is a derived edge to another node computed from more primitive ordering constraints. (Since domination depends non-locally on edges of nodes between the two nodes in question, it is never simply an independently editable attribute of a single node.)
    - A node may be decorated with a sequence number after a pass computing reverse post-order. (*RPO* numbering is not independently editable, and can only be computed by a pass over the whole program.)
    - A constraint node may be created (but later removed) to represent a type constraint that is discovered to hold within a subgraph.
- Nodes and edges include **decorations** conveying additional information about the program.
- Since decorations are assertions about nodes, they are distinct from nodes.

IR types

- **IR-specific types** are an important decoration on nodes. (We usually just call them "IR types" or "types", remembering that they differ from types in the source and implementation languages.)
- A type represents a set of reliable assertions about a node (value or perhaps other behavior). (Equivalently, a type can be viewed as representing a set of possible concrete values that a node may produce.)
- Types are distinct from source-level data types and declarations, although they typically refer to them or embed them.
- Implementation data types are distinct from IR types, although they often refer to them.

- An **IR type system** is a formal scheme of possible types which model specific sets of assertions or values.
- There is no maximal, universal, or perfectly natural type system that can be expressed in practical limits.
- Nevertheless, the following IR types are known to be useful:
    - discrete values, including null and integral ranges
    - the type of a value that is indeterminate except for its machine-level base type
    - the type of a value which is impossible or paradoxical to compute
    - the type of a reference value, as its source-level class (plus an indication of other discrete values, such as null)
- As there is some utility in defining that every node has a type, there may also be a requirement for that all nodes have an assignable type, including control flow and memory effect nodes.

Control flow

- A node's **continuation** is the selection of which part of the program to execute next.
- A continuation may be normal or **exceptional**, in the Java sense.
- A continuation may be normal or **uncommon**, in the sense of optimistic optimizations with slow paths.
- A node with only continuations and/or effects can be viewed as having a "void" value, mathematically a unit.
- A node with an **empty** continuation is also possible; this would be an error or halt that exits the program.
- A node which depends on an empty continuation is not reachable, and can be safely deleted from the graph.
- A node `B` directly related to a continuation of a node `A` is a successor of `A`, and `A` is a predecessor of `B`. (Note that nodes do not necessarily have explicit successors or predecessors.)
- A **floating node** has no explicit predecessor or successor, but is scheduled by examining its other dependencies.
- A **pinned node** is a node which is explicitly contained in an associated control flow construct. In generated native code, the pinned node will be in the same basic block as its control flow construct.
- Control can both fork and join (split and merge). Any IR will have conventions for representing this.
    - If the IR has explicit control flow joins (merges), these will be represented as nodes, such as C2 `RegionNodes`.

Effects

- A node's "effects" are descriptions which memory locations might have been changed after a node executes.
- An "effect system" is a formal scheme of possible memory locations that a program might create or change during its execution.

- Both continuations and effects can be viewed as values, if and when this is convenient.
- In that case, the effect system could be part of the type system.

Profiles

- Profiles are an important decoration on nodes.
- A profile represents a set of speculative **assertions** about a node (value or other behavior).
- Profile data is gathered from execution of previous instances of the code, or perhaps from other analyses.
- Profile data may include estimates of the likelihood of a node and/or its continuations being executed.
- Profile data may include estimates of the type of a node's value.

Containment

- In order to express locality in the program, a node may have a many-to-one edge relation to a "container" node.
- Containers express methods, scopes, loops, basic blocks, or other units of placement.
- Containers are special-purpose nodes, but if the containment is compositional, they can be treated the same as other nodes.
- Alternatively, containment can be expressed using decorations and perhaps multiple (disconnected) graphs.

Implementation

- As a matter of both design and engineering, edges and decorations can be represented directly or implicitly.
- A direct edge or decoration is independently determined for each node.
- An implicit edge or decoration is derived by some rule from a node's neighbors and decorations.
- Containment, continuation, and effects are typically implicit, in part, as a way of reducing edge complexity.
- Values are usually not implicit, but can be implicit in stack-oriented notations, where a value is accessed via a LIFO stack implicitly used by operations (like bytecode `iadd`).

CFG vs. PDG

- A graph must eventually be **scheduled**; that is, it must be given basic block assignments plus a total order consistent with its causality constraints. (This is because machine instructions necessarily occur within basic blocks, and have a total order therein.)
- A **CFG** (or "control flow graph") version of an IR is one in which each node is pinned to a basic block and (usually) scheduled within its basic block.
- A CFG IR is can always be easily scheduled, under the assumption that no node is pinned to an inconsistent basic block.

- In traditional CFGs, control placement is specially represented, by including each node within a sequence representing all instructions in a common basic block.
- A **PDG** (or "program dependence graph") represents all control by explicit edges.
- In a PDG, nodes get scheduled using **GCM** ("global code motion") or a similar scheduling algorithm, but only when a placement is needed.
- In a PDG, the normal state of a typical node is to float relative to its control edges.
- When processing a PDG, a temporary CFG may be constructed (say for loop transforms) as a temporary side-structure, not as a relation intrinsic to the IR graph.
- The placements in a **CFG** may be efficiently determined when the IR is parsed from a totally ordered representation, such as an AST or bytecodes. In this case, we may say that nodes are **pre-scheduled**.
- In a CFG, even the simplest change to placement requires editing passes and temporary reification of control edges. For that reason it is desirable to make a good placement at parse time (or other graph creation time). This typically works well because the source encounter order approximates a good schedule.
- Debatably, in a PDG the explicit edges consume more working storage than the implicit ones of a CFG. In the PDG case, the order of nodes in memory is immaterial and can be considered "wasted" information. On the other hand, memory order (if significant) is a rigid relation which easily over-specifies order beyond the true, essential constraints, making it more difficult to perform reordering transforms.
- Bugs come in at different places in the two approaches.
    - With PDG, a mis-edited edge can cause errors in the next schedule, which might be long after the bad edit.
    - With CFG, a misplaced node can (perhaps) be more easily checked against its neighbors, because of the excess of (implicit) connections to them.
    - But with CFG it is easier for node placement to be "correct by accident", due to original presentation from source, and this can lead to low-frequency bugs from unrelated transforms, which are hard to diagnose.
- Engineers find it easy to read sequential presentations of CFG basic blocks, but usually cannot read PDG graphs directly without assistance; this affects debugging speed. (This is why easy inspection is a requirement, regardless of IR details.)
- In a PDG, the concentration is on free but local relations. Every edge is necessary and sufficient to the program semantics as a whole. When this is true, we maximize ease and robustness of local analysis and editing, which are not the whole optimization story but a major part of it.
- Storage overheads of explicit control edges in a PDG are probably not inherently significant. They can be controlled by engineering tricks (as with annotations in Graal).
- Even in a PDG, debugging presentations can easily include trial schedules to be competitive with CFG presentations.
- The PDG format may be regarded as an earlier or higher-level representation, which is eventually lowered to a CFG as part of the process of generating machine code.