Abstract Interpretation

and

Low Level Code Optimization

Saumya K. Debray

University of Arizona
Tucson, AZ 85721

Program Optimization

Need to know:

1. When an optimization can be applied.
   - Optimizations must preserve program semantics.
     "Bad things" should not happen.
   - Cannot be applied until the appropriate preconditions are met.
2. When an optimization should be applied.
   - Often, optimizations involve a tradeoff, e.g.:
     – space for speed; or
     – speed in one part of a program for speed in another part.
   - Code optimization should improve program performance (or, at least, not hurt it).

Why Semantics-Based Analysis Techniques?

- Establish precise relationships with the formal semantics of the language.
- Obtain insights into the analysis process and develop alternative analysis techniques.
  Example: widening/narrowing in abstract interpretation.
- Develop frameworks that allow the reuse of meta-level results.

High Level vs. Low Level Optimizations

High Level :

– expressible in source level terms;

– can be effected (not necessarily conveniently) by the programmer;

– Examples: recursion removal, strength reduction, deforestation.

Low Level :

– not expressible in source level terms;

⋆ – cannot be effected by the programmer;

⋆ – can produce significant performance improvements.

– Examples: register allocation, instruction scheduling.

Low Level Optimizations: An Example

- jc : a compiler for a (dynamically typed) logic programming language:

  compiles to C;

  optimizations:

  - various low-level control flow optimizations;
  - some inter-procedural “register” allocation;
  - unboxing floating point values.

- Low level optimizations produce
  - a 2× speed improvement (3.4× if we count last call optimization)
  - significant improvements in heap usage

- Compared to C code written in a natural style:
  - ∼ 13% slower than `gcc -O2`;
  - ∼ 25% faster than `cc -O2`;
  - ∼ 6% faster than `cc -O4`.

Semantic Levels in Program Analysis

- Semantics-based analyses typically defined w.r.t. some “concrete” semantics.

- For low level concerns, the concrete semantics is usually not concrete enough!

  Implementation level details, e.g., registers, pointers, etc., are abstracted away to obtain the concrete semantics.

- *Obvious Idea* : Use a low level concrete semantics.

  Issue : Representation of the program for analysis purposes:

  - high level, e.g.: abstract syntax tree
  - low level, e.g.: virtual machine code.

Using High Level Representations I

Use high level representation to reason directly about low level matters.

Problem : “Impedance mismatch” :

- Ensuring consistency between high level analysis and low level processing [code generation, optimization].

  Example : Aquarius Prolog [Van Roy 1990] :
  Pointer dereferencing optimization requires explicit checks for consistency in code generator.

- Anticipating effects of other transformations and low level decisions.

  Example : Orbit Scheme compiler [Krantz 88] :
  Interactions between closure analysis and register allocation become complicated.

Using High Level Representations II

“Promote” low level properties to a form amenable for high-level manipulation.

Example : Boxing/unboxing analysis using “representation types” [Peyton Jones & Launchbury 91, Leroy 92] :

- Expose low level representation (boxed/unboxed) by enriching the type system.
- Representation optimization can then be formulated in source level terms.
- Use type coercion to effect representation conversions where necessary.

Limitation : Not all properties amenable to this kind of promotion.

Example : Compile time reference counting for destructive aggregate updating.

Using Low Level Representations

Analyze low level representation (e.g., a sequence of virtual machine instructions) using a low level concrete semantics.

Issues :

1. *Performance* : analysis may be less efficient:
   - larger size of low level representations
   - more low level entities (registers etc.) and operations
2. Precision : information about relationships between objects may be harder to reconstruct.

Reducing the Cost of Low Level Analysis

Source Program "Concrete" Meaning

$$\begin{tikzcd} P \arrow[r, "semantics"] \arrow[d, "\substack{transformation \\ \tau}"'] & M \arrow[d, "\substack{abstraction \\ \alpha}"] \\ P' \arrow[r, "semantics"] & M' \end{tikzcd}$$

"Approximate" Program "Abstract" Meaning

Basic Idea : specialize abstract interpreter to the input program P (+ abstract domain).

Implementation : as a program transformation:

- generate P' in one pass over P;
- execute P' to get the results of analysis.

Analysis of Low Level Representation

- Usual approach to analysis [Le Charlier, Van Hentenryck, *et al.*; Van Roy; Taylor; . . .] :

  Interpret the program (in abstract syntax tree form) over the abstract domain.

- Abstract compilation [Debray 87; Hermenegildo *et al.* 92] :

  Compile the input program to obtain a Prolog program which, when executed, will produce the analysis results.

- This idea can be extended logically to use a low level representation [Tan & Lin 92, Krall & Berger 95]:

  Compile the input program to obtain a sequence of WAM instructions (in byte-code form) that can be executed (on a special "abstract WAM engine.") to produce the analysis results.

- Each step in this progression reduces the cost of various abstract operations.

Analysis of Low Level Representation: *Advantages*

- "Impedance mismatch" between high level representation (syntax tree) and low level information (registers etc.) is avoided for low level code optimizations.

- Code optimizations that change the program can be accommodated easily.

- Different abstract domains can be used for different analyses by changing the "abstract WAM engine."

  The Prolog-to-WAM compiler is unaffected.

- Speed: Tan & Lin report speedups of 57–575 (average speedup: 152) compared to Van Roy's Aquarius system.

Inferring Relationships between Objects

- Relationships easily discovered at a high level may require complicated (possibly imprecise) pointer chasing at low level.
- Not all structural relationships may be amenable to high-level analysis.

  E.g.: sharing relationships may depend on specific implementation decisions invisible at a high level.
- *A Possible Solution* : Combine high-level and low-level analyses:
  - Annotate high-level program representation with information from high-level analyses.
  - When translating the program to lower level representations, translate also the associated analysis information.

Using Analysis Results for Optimization

- Usually, analyses say whether an optimization *can* be applied, not whether it *should*.
- *"Bad things should not happen (due to compiler optimizations)."*

  Usually, "bad things" $\cong$ semantic incorrectness.
- However, a (significant) deterioration in performance (space, time, . . .) is arguably a Bad Thing.
- Without careful attention to low level details, even plausible "optimizations" can go awry.

Optimizations that may Pessimize 1

Subroutine Inlining (a.k.a. unfolding) :

Motivation : eliminate subroutine call/return overhead.

Pitfalls :

- Can cause a slowdown due to register usage effects [Davidson & Holler 92].
- Can increase compile time memory requirements significantly [Richardson 91].
- Can increase runtime memory requirements due to increase in stack frame size.

  (cf. Safety considerations [Chase 88].)

Optimizations that may Pessimize 2

Closure Allocation Optimization [Krantz 88]

Motivation : allocation/deallocation may be faster for stack than for heap.

Pitfalls :

- Stack allocation of closures can lead to increased memory requirements.

  (cf. Safety considerations [Chase 88].)

Optimizations that may Pessimize 3

Representation Optimization for Boxed Values

Motivation : reduce space (for boxing) and time (for boxing/unboxing) requirements in polymorphic and dynamically typed languages.

Pitfalls :

- Unwise placement of representation conversion operations can cause a slowdown in execution speed.
  (cf. [Peterson 89] for a treatment that takes execution frequencies and representation conversion costs.)

Optimizations that may Pessimize 4

Memoization

Motivation : avoid repeating the same computation.

Pitfalls :

- The overhead of table management can overwhelm the benefits of avoiding recomputations, leading to a slowdown.
- The memo tables can increase the overall storage requirements of a program.

Code Optimization and Performance Improvement

- Often, an optimization involves a cost reduction in one place for a cost increase elsewhere.
- Low level costs of different operations can be very different.
  E.g.: cost of boxing a value is generally more than that of unboxing it.
- Need low level cost information as well as execution frequency estimates to determine *whether* an optimization should be applied.
  Examples :
  - Inlining: Chang et al. 92, Dean&Chambers 94.
  - Representation optimization: Peterson 89.
- Can we say anything formally about these parameters?

Execution Frequency Information: Practice

Structural Heuristics :
E.g.: estimates based on loop nesting depth.

(+) : simple.

(−) : May not relate to reality;
Typically, beyond the user's control.

Profile-based :

(+) : Precise [if inputs well chosen].

(−) : Choosing "typical" inputs may not be easy.

Overall :

(+) : Results directly applicable to low level code optimization;
Tried, seem to work (sometimes).

(−) : Ad hoc;
Not amenable to formal reasoning.

Execution Frequency Information: Theory

- A lot of work on estimating the computational complexity of programs:

  Algol : Cohen & Zuckerman 74;

  Functional Langs : Wegbreit 75, Schultis 85, Le Métayer 88, Bjerner & Holmström 89, Sands 90;

  Cost Systems : Gifford *et al.* 92, 94;

  Prolog : Lin & Debray 93.

- Costs typically obtained as functions of input size:

$$Cost_f(n) = 0.5n^2 + 1.5n + 1$$

- Overall :

(+) : Grounded in semantics;
Amenable to formal reasoning.

(−) : Not directly suitable for low level code optimization.

Possible Reconciliation

- Use semantics-based cost functions to obtain thresholds where particular optimizations become applicable.

- Use polyvariant specialization at a low level [basic blocks?] to handle the different cases that arise.

- E.g. : Dynamic Parallelization [Debray *et al.* 90, Huelsbergen *et al.* 94, Reistad & Gifford 94.] :
  - Create two versions of potentially parallel code, for sequential and parallel execution.
  - Dynamically choose the appropriate version based on input sizes.

- Potential Problems :
  - combinatorics of the analysis
  - code bloat

Summary

- Typical semantics-based analyses use concrete semantics that are "too abstract."

  Low level information is not retained.

- Determining when an optimization can be applied:
  - Choice of representation.
  - How can performance issues be addressed?

- Discovery of relationships between objects in low-level analyses.

- Determining when an optimization should be applied:
  - Estimating execution frequencies.
  - Combinatorial issues in low-level polyvariant specialization.