Compiler Optimization Modulo Axioms

Eivind Jahren
Anya Helene Bagge

with the Bergen Language Design Laboratory (BLDL)

Department of Informatics
Bergen University

Rajeev Joshi, Greg Nelson, and Keith Randall.
Denali: a goal-directed superoptimizer.
2002.

David Lacey and Oege De Moor.
Imperative program transformation by rewriting.
In *Compiler Construction*, pages 52–68. Springer, 2001.

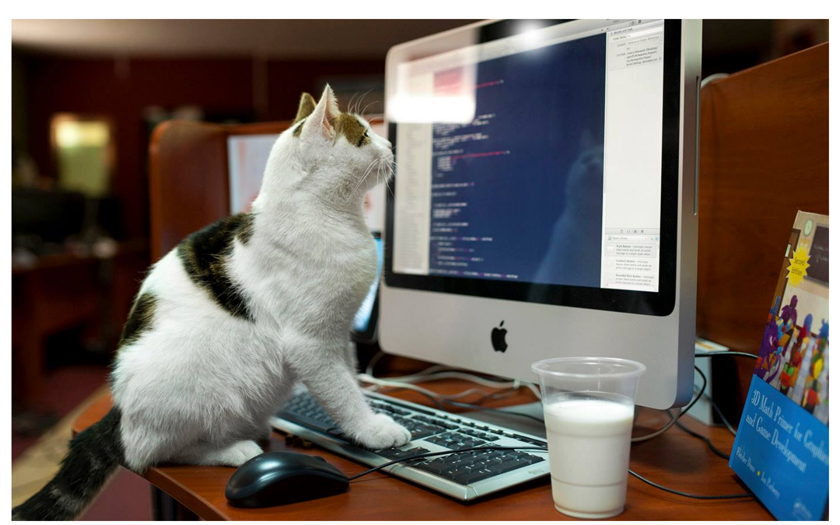
Xiaolong Tang and Jaakko Järvi.
Concept-based optimization.
Proceedings of the 2007 Symposium on Library-Centric Software Design - LCSD '07, pages 97–108, 2007.

Introduction

- Optimization often uses hardcoded information.
- Programmers abstraction is not visible to the compiler.
- Information used for verification can often be used for optimization.
- Several techniques (such as SSA form) apply to both.

Introduction

- Optimization often uses hardcoded information.
- Programmers abstraction is not visible to the compiler.
- Information used for verification can often be used for optimization.
- Several techniques (such as SSA form) apply to both.
- Goal: Incorporating verification techniques into compiler optimization.
- Poses an additional challange: Only exhaustive checks are useful.


and Game Development

Assert $0 \leq \text{i} < \text{array.length}$;
b := array[i];

Example

```
deltax := x1 - x0;
deltay := y1 - y0;
error := deltax / 2;
y := y0;
for x from x0 to x1 do
    safeplot(x,y) ; // performs a bounds check
    if error < 0 then
        y := y + 1;
        error := error + deltax;
    end
end
```

Example

```
deltax := x1 - x0;
deltay := y1 - y0;
Assert((deltax > 0) and (deltay > 0) and deltay ≤ deltax);
error := deltax / 2;
y := y0;
for x from x0 to x1 do
    Invariant(x0 ≤ x ≤ x1);
    Invariant(y0 ≤ x ≤ y1);
    Invariant(...);
    safeplot(x,y) ; // performs a bounds check
    if error < 0 then
        y := y + 1;
        error := error + deltax;
    end
end
```

Example

```
deltax := x1 - x0;
deltay := y1 - y0;
Assert((deltax > 0) and (deltay > 0) and deltay ≤ deltax);
error := deltax / 2;
y := y0;
for x from x0 to x1 do
    Invariant(x0 ≤ x ≤ x1);
    Invariant(y0 ≤ x ≤ y1);
    Invariant(...);
    unsafeplot(x,y) ;
    if error < 0 then
        y := y + 1;
        error := error + deltax;
    end
end
```

- Liveness.
- Common subexpression elimination.
- Rewriting rules of the form $\forall \mathbf{a}.P(\mathbf{a}) \Rightarrow E_1(\mathbf{a}) \to E_2(\mathbf{a})$.
  eg. $\forall x_1, x_2.x_2 = 0 \Rightarrow x_1 + x_2 \to x_1$

$\forall \mathbf{a}.P(\mathbf{a}) \Rightarrow E_1(\mathbf{a}) \rightarrow E_2(\mathbf{a})$.
Use unification to determine whether the rewriting rule is applicable. Use theorem prover to determine whether the guard holds.
Not as strong as $(\forall \mathbf{x}.E_1(\mathbf{x}) = E_2(\mathbf{x})) \models \exists \mathbf{z}.E_2(\mathbf{z}) = Q(\mathbf{y})$.

- Do away with data dependency with SSA form.
    - Only one assignment for each variable.
    - Data dependencies hidden behind phi functions.
- Do away with control dependency with post domination front.
- Does not capture (all) interdependencies between data and control.

Example

```
deltax := x1 - x0;
deltay := y1 - y0;
Assert((deltax > 0) and (deltay > 0) and deltay ≤ deltax);
error_1 := deltax / 2;
y_1 := y0;
for x from x0 to x1 do
    y_3 := φ(y_1,y_2);
    error_3 := φ(error_1,error_2);
    safeplot(x,y_3) ; // performs a bounds check
    if error_3 < 0 then
        y_2 := y_3 + 1;
        error_2 := error_3 + deltax;
    end
end
```

Future Work

- Experimental evaluation.
- Experiment with Candidate annotation generation.