Using types to optimize programs

Amer Diwan

The Question

Are type declarations in statically typed languages useful for optimizing programs?

Disadvantage: Too imprecise. Or are they?
Advantages:
– Fast
– Do not require the whole program
Assumption: Object-oriented programming language
Outline

• Using types to do pointer analysis
  – this class
  – three algorithms
  – evaluation to see how well they work
• Using types to resolve method invocations
  – next class
  – five algorithms
  – evaluation to see how well they work

Using types for pointer analysis
A Running Example

Type T

- T *t
  - f: INT
  - g: INT

Type S1

- S1 *s1

Type S2

- S2 *s2

t, s1, and s2 are references to memory locations in the heap

Running Example (cont.)

- ...:= t->g;
- ...:= s1->f;
  - s2->f :=...
- ...:= s1->f;
  - s2->f :=...
- ...:= s1.f

Redundant load elimination eliminates lexically identical redundant heap references
Analysis I: TypeDecl

Use type compatibility only:
\[
\text{TypeDecl}(p, q) = \text{Subtypes}(\text{Type}(p)) \cap \text{Subtypes}(\text{Type}(q)) \neq 0
\]

Analysis II: FieldTypeDecl

Use other properties of types, e.g.:
- Accesses to distinct fields cannot alias each other
- An array reference cannot alias a field reference
- Must consider subpaths, pass by reference, ...

\[
(t, s_1) \\
(t, s_2) \\
(t.f, t.g) \\
\]

\[
(t, s_1) \\
(t, s_2) \\
(t.f, t.g) \\
\]

\[
\]

\[
\]

T
S1  S2

T
S1  S2

Analysis III: SMFieldTypeDecl

- Incorporate flow insensitive analysis. t aliases s1 if
  - at some point, a reference to an object of type S1 may have been assigned to a location of type T (S1 is merged into T)

\[ (t, s1) \]
\[ (t', s) \]
\[ (t', s, g) \]
\[ (s, s') \]

E.g., \( t = \text{new } S1; \)

Evaluation Environment
Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines</th>
<th>Dynamic heap loads (% total instrs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>format</td>
<td>395</td>
<td>10</td>
</tr>
<tr>
<td>dformat</td>
<td>602</td>
<td>9</td>
</tr>
<tr>
<td>write-pickle</td>
<td>654</td>
<td>13</td>
</tr>
<tr>
<td>k-tree</td>
<td>726</td>
<td>10</td>
</tr>
<tr>
<td>slisp</td>
<td>1,645</td>
<td>27</td>
</tr>
<tr>
<td>m2tom3</td>
<td>10,574</td>
<td>8</td>
</tr>
<tr>
<td>m3cg</td>
<td>16,475</td>
<td>8</td>
</tr>
</tbody>
</table>

Static Evaluation

Measure **alias pairs**.

E.g., \((p,q) \equiv p\) and \(q\) are references in the program that *may* reference the same heap location.

⇒ **Enables comparing analyses**

What it does **not** do:

- Allow us to compare analyses with different strengths
- Tell us how effective the analysis is w.r.t. clients
- Tell us how much better we could do
Static evaluation

Alias pairs within procedure as a percent of all possible pairs within procedure

<table>
<thead>
<tr>
<th></th>
<th>TypeDecl</th>
<th>FieldTypeDecl</th>
<th>SMFieldTypeDecl</th>
</tr>
</thead>
<tbody>
<tr>
<td>format</td>
<td>31</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>dformat</td>
<td>24</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>write-pickle</td>
<td>24</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>k-tree</td>
<td>29</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>slisp</td>
<td>45</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>m2tom3</td>
<td>41</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>m3cg</td>
<td>32</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

• These numbers look pretty bad by themselves!
• FieldTypeDecl better than TypeDecl
• SMFieldTypeDecl doesn’t offer much

Dynamic Evaluation

Measure run-time impact of RLE

⇒ Directly measures impact of an analysis on its clients

What it does not do:
  – Give results for all inputs and optimizations
  – Tell us how much better we could do
Run-time improvements with RLE

Limit Evaluation

Measure upper-bounds on performance: count heap references that are still redundant after redundant load elimination.

⇒ Reveals potential room for improvement

What it does not do:
- Give results for all inputs and optimizations
## Limit Evaluation (intraprocedural): Loads that are still redundant

<table>
<thead>
<tr>
<th>Format</th>
<th>Redundant originally</th>
<th>Redundant after RLE and SMFieldTypeDecl</th>
</tr>
</thead>
<tbody>
<tr>
<td>format</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>dformat</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>write-pickle</td>
<td>14</td>
<td>5</td>
</tr>
<tr>
<td>ktree</td>
<td>34</td>
<td>21</td>
</tr>
<tr>
<td>slisp</td>
<td>56</td>
<td>16</td>
</tr>
<tr>
<td>m2tom3</td>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td>m3cg</td>
<td>22</td>
<td>3</td>
</tr>
</tbody>
</table>

**Why still redundant?**

- Hidden loads
- No PRE
- No copy prop.
- Rest

"Rest" is the upper-bound on what a better alias analysis can do.
Summary of types for pointer analysis

- Despite large number of alias pairs, type-based alias analysis is nearly perfect for our benchmarks and optimizations
- More precise analysis is not necessarily better
- The three evaluation techniques tell us different things and should all be used.
- Type-safety can be used to improve program performance!

Discussion

- **Strengths**
  - Simple and fast analysis that works for an important application
- **Weaknesses**
  - Doesn't work for unsafe languages
  - Paper tells us type-based pointer analysis works well for 2 uses of pointer analysis. What about other uses?
Using types for resolving method invocations

The problem

- Method invocations in object-oriented languages degrade performance
  - directly since dynamic dispatch takes more time than direct calls
  - indirectly by diluting control flow information and thus inhibiting compiler optimizations

Can we use types to replace method invocations by direct calls?
Definitions

• A **polymorphic** method invocation site calls more than one procedure at run time
• A **monomorphic** method invocation site always calls the same procedure at run time
• **Resolving** a method invocation site identifies it as monomorphic

Full resolution is **undecidable** so any resolution technique must be conservative: assume polymorphic

---

Analysis 1: type hierarchy analysis

• **Bounds the procedures a method invocation may call by examining the type hierarchy declaration for method overrides**

```
T
  m

S
  m
  n
```

t: T; …; t->m() can call? T::m or S::m
s: S; …; s->m() can call? S::m
Analysis 2: intraprocedural type propagation

- Propagate types along control flow edges. Assume worst case for calls and pointer assignments.

```
T *o; T *p;
p = new S;
op->m();
o->m();
o = p;
o->m();
op->m();
o = new T;
o->m();
```

Analysis 3: intraprocedural type propagation + tbaa

- Intraprocedural type propagation, but uses tbaa to disambiguate pointer dereferences.

```
T *s1; S2 ** s2;
s2 = …;
s1 = new S1;
*s2 = new S2;
s1->m();
```

**Without tbaa**
calls S1::m or S2::m

**With tbaa**
calls S1::m
Analysis 4: interprocedural type propagation

• Similar to intraprocedural type propagation
  – includes monovariant analysis of calls

Analysis 5: interprocedural type propagation + tbaa

• Simple extension of interprocedural type propagation with tbaa
Effectiveness of analyses

Percentage of total method invocations

- Ignoring NIL
- With NIL

<table>
<thead>
<tr>
<th>Method</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>tba</td>
<td>0.00%</td>
</tr>
<tr>
<td>tpa</td>
<td>165.747%</td>
</tr>
<tr>
<td>tpa+iba</td>
<td>165.747%</td>
</tr>
<tr>
<td>tpa+ip</td>
<td>461.748%</td>
</tr>
<tr>
<td>tpa+ip+iba</td>
<td>461.1100%</td>
</tr>
</tbody>
</table>

How many monomorphic are resolved?

Percentage of total invocations

- Unresolved
- Resolved

<table>
<thead>
<tr>
<th>Library</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>format</td>
<td>(65.0)</td>
</tr>
<tr>
<td>dformat</td>
<td>(94.0)</td>
</tr>
<tr>
<td>write-pickle</td>
<td>(128.6)</td>
</tr>
<tr>
<td>slisp</td>
<td>(2.5)</td>
</tr>
<tr>
<td>k-tree</td>
<td>(494.16)</td>
</tr>
<tr>
<td>postcard</td>
<td>(155.56)</td>
</tr>
<tr>
<td>m2tom3</td>
<td>(8.12)</td>
</tr>
<tr>
<td>m55</td>
<td></td>
</tr>
<tr>
<td>treacle</td>
<td></td>
</tr>
</tbody>
</table>
Summary

- Simple analyses can resolve many method invocations
  - Tbaa helps method resolution
  - There is little room for improvement in these analyses for the benchmarks
Discussion

• Strengths
  – Simple and apparently effective analyses
• Weaknesses
  – Assumes type safety
  – ?

Next topic

• Closures
• Reading: read about activation records, static and dynamic links in your favorite programming languages text