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:

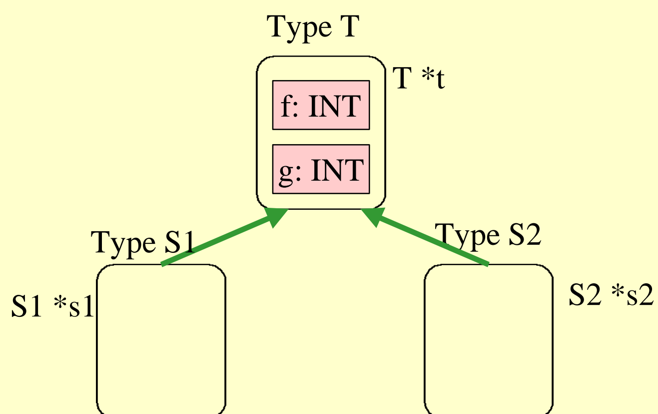
- Fast
- Do not require the whole program

Assumption: Object-oriented programming language

Outline

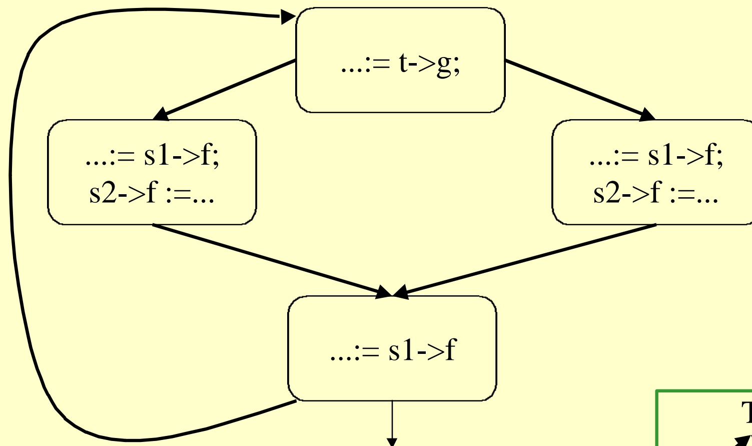
- Using types to do pointer analysis
  - this class
- Using pointer analysis to improve program performance
  - next class
  - evaluation to see how well they work

A Running Example

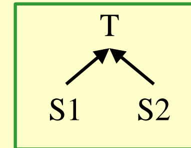


t, s1, and s2 are pointers to memory locations in the heap

Running Example (cont.)

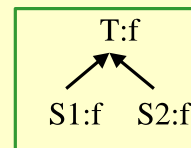


Redundant load elimination eliminates lexically identical redundant heap references



Another example

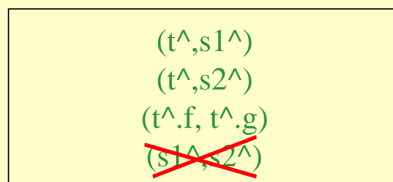
- t = NEW(s1)
  x^ = NEW(s2)
  t.f();
- What does t.f() call?

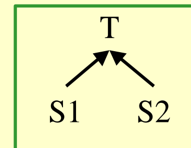


Analysis I: T-TBAA

Use type compatibility only:

T-TBAA(p, q) =
 $Subtypes(Type(p)) \cap Subtypes(Type(q)) \neq 0$





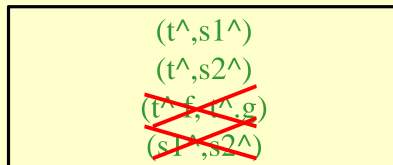
Weaknesses of T-TBAA

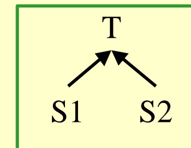
- Ignores some type information
  - E.g., field names could also be used
  - TF-TBAA
- Ignores instructions in the program: only considers type declarations
  - TFM-TBAA
  - TM-TBAA

Analysis II: TF-TBAA

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 by reference, ...

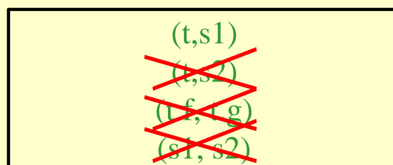


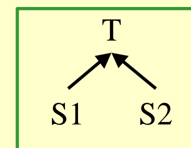


Analysis III: TFM-TBAA

- 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)

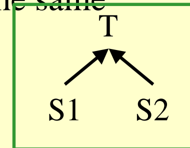
e.g., t = new S1;





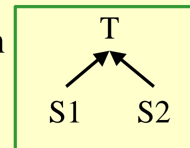
Qualitative analysis

- How do TBAA compare to Steensgaard in precision?
  - t1: T; t2: T;
    t1 = new T;
    t2 = new T;
  - According to Steensgaard t1 and t2 do not point to the same object
    According to TBAA they may point to the same object



Qualitative analysis (cont)

- s1: S1; s2: S2; t: T;
  s1 = new S1; s2: new S2;
  t = s1; t = s2;
- According to Steensgaard, t, s1, and s2 may all point to the same object
- According to TBAA, s1 and s2 may not point to the same object
- Bottom line: incomparable w.r.t. precision



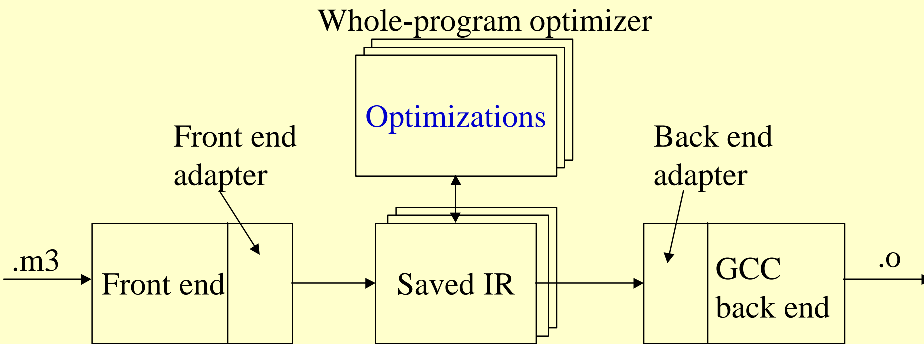
When will TBAA work best?

- (When programs are written in a safe language)
- When programs use types carefully rather than having everything by a void* or ROOT
  - t: T; s1: T; s2: T versus
    t: T; s1: S1; s2: S2
- How well does it work for real programs?

Now that I have a pointer analysis, what do I do with it?

- Eliminate redundant loads
- Replace method invocations by direct calls
  - Type hierarchy analysis
  - Intra and Interprocedural type propagation
  - ...with and without TBAA

Evaluation Environment



Some of the benchmarks

Benchmark	Lines	Dynamic heap loads (% total instrs)
format	395	10
dformat	602	9
write-pickle	654	13
k-tree	726	10
slisp	1,645	27
m2tom3	10,574	8
m3cg	16,475	8
trestle	28,977	

Static Evaluation

Measure alias pairs.

E.g., (p,q) ≡ 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

	T-TBAA	TF-TBAA	TFM-TBAA
format	31	27	27
dformat	24	16	16
write-pickle	24	13	13
k-tree	29	17	17
slisp	45	33	33
m2tom3	41	23	23
m3cg	32	5	5
trestle	23	11	11

- These numbers look pretty bad by themselves!
- TF-TBAA better than T-TBAA
- TFM-TBAA doesn't offer much

Dynamic Evaluation

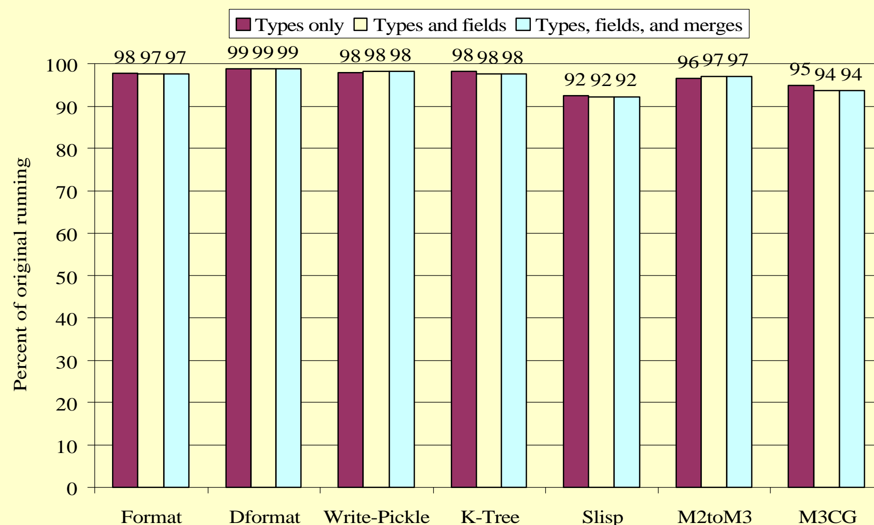
Measure run-time impact of RLE and method resolution

⇒ Directly measures impact of an analysis on its clients

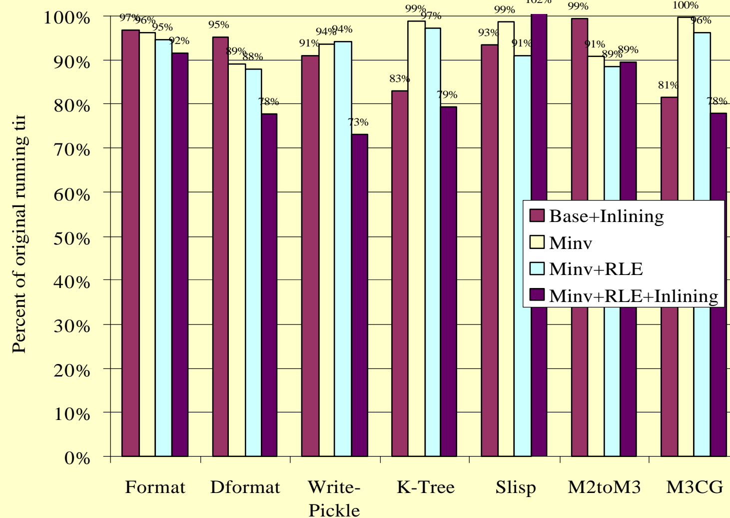
What it does not do:

- Give results for all inputs and optimizations
- Tell us how much better we could do

Run-time improvements with RLE



Overall run-time improvement



Limit Evaluation

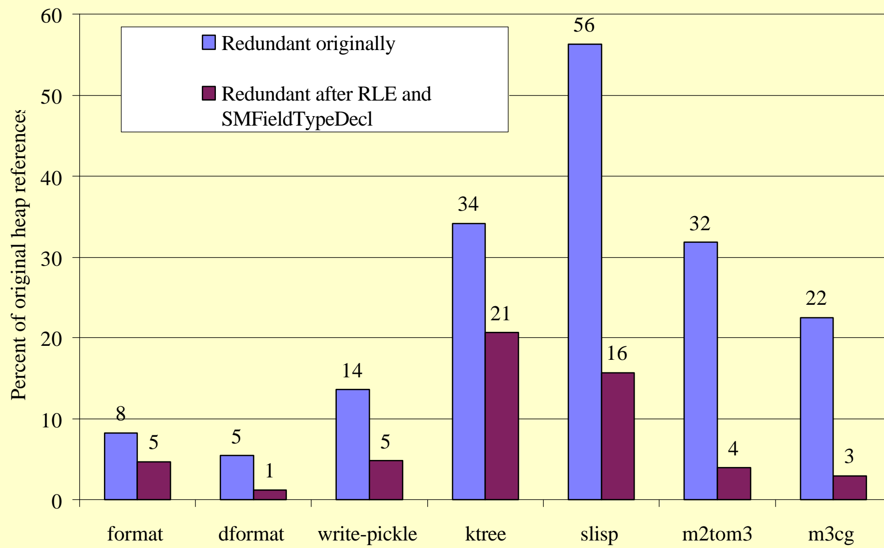
Measure upper-bounds on performance:

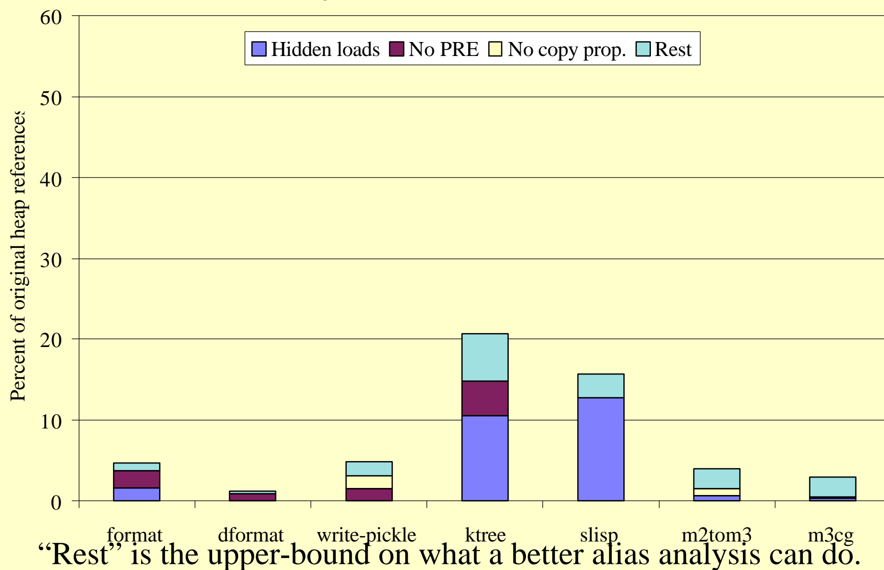
- count heap references that are still redundant after redundant load elimination.
- count methods that are still unresolved but call the same procedure at run time

⇒ Reveals potential room for improvement

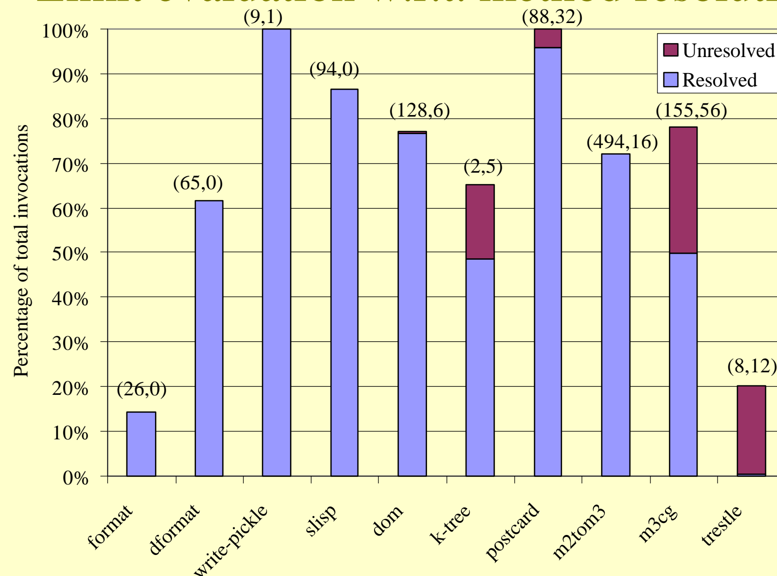
What it does not do:

- Give results for all inputs and optimizations


Limit Evaluation (intraprocedural):
Loads that are still redundant
Redundant originally
Redundant after RLE and SMFieldTypeDecl
Percent of original heap references
0
10
20
30
40
50
60
8
5
5
1
14
5
34
21
56
16
32
4
22
3
format
dformat
write-pickle
ktree
slisp
m2tom3
m3cg


Why still redundant?
Hidden loads
No PRE
No copy prop.
Rest
Percent of original heap references
0
10
20
30
40
50
60
format
dformat
write-pickle
ktree
slisp
m2tom3
m3cg
"Rest" is the upper-bound on what a better alias analysis can do.

Limit evaluation w.r.t. method resolution



Summary of types for pointer analysis

- Despite large number of alias pairs, type-based alias analysis is nearly perfect for these 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
  - How well do TBA work for other uses of pointer analyis?

Next topic

- Closures
- Reading: Scott 3.3