

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

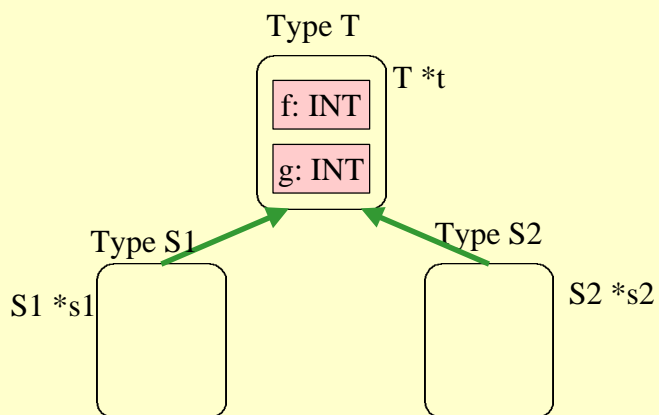
Assumption: Object-oriented programming language

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

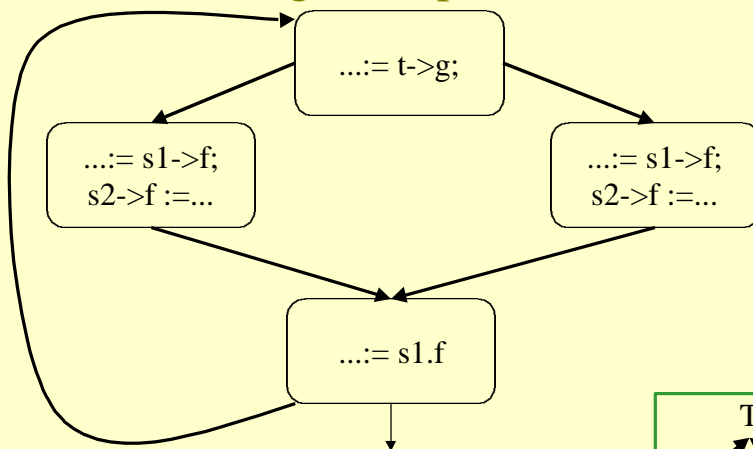
Using types for pointer analysis

A Running Example

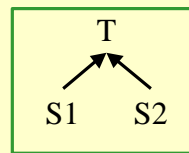


t, s1, and s2 are references to memory locations in the [heap](#)

Running Example (cont.)



Redundant load elimination eliminates lexically identical redundant heap references

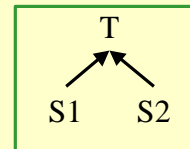
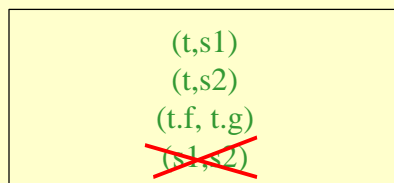


Analysis I: TypeDecl

Use type compatibility only:

TypeDecl(p, q) =

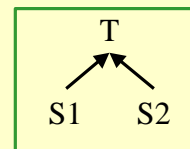
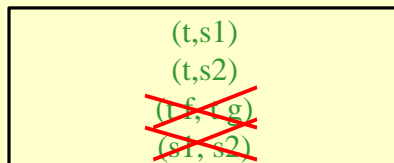
$\text{Subtypes}(\text{Type}(p)) \cap \text{Subtypes}(\text{Type}(q)) \neq 0$



Analysis II: FieldTypeDecl

Use other properties of types, e.g.,

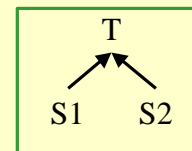
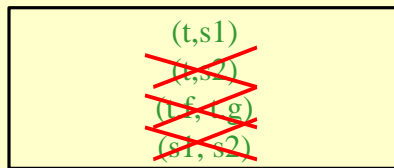
- Accesses to distinct fields cannot alias each other
- An array reference cannot alias a field reference
- Must consider subpaths, pass by reference, ...



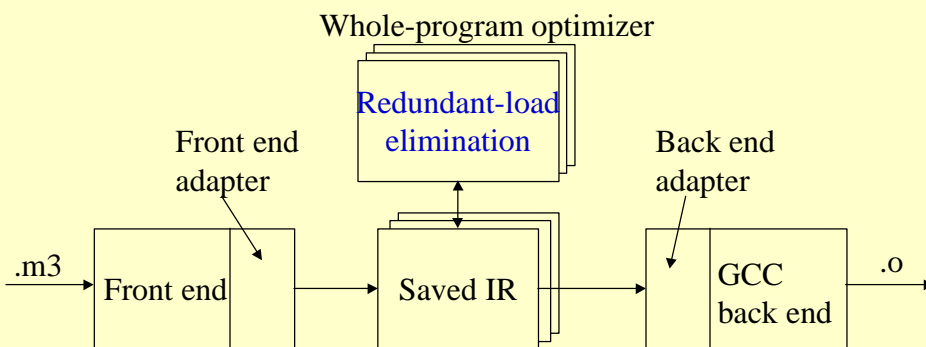
Analysis III: SMFieldTypeDecl

- 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)

e.g., $t = \text{new } S1;$



Evaluation Environment



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

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

	TypeDecl	FieldTypeDecl	SMFieldTypeDecl
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

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

Dynamic Evaluation

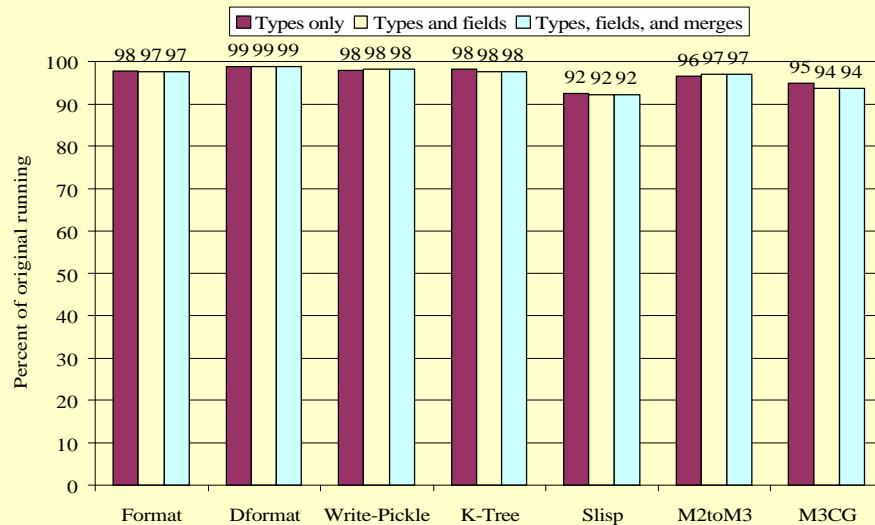
Measure run-time impact of RLE

⇒ 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

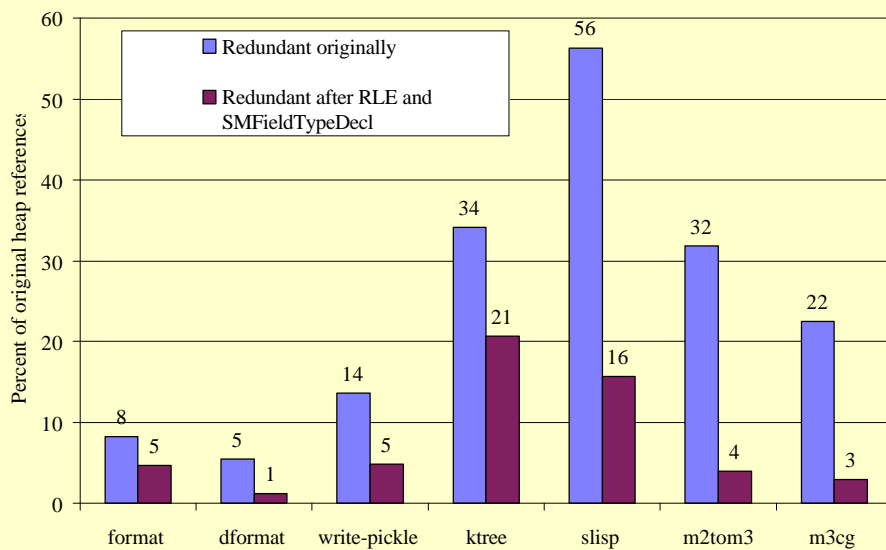
Measure upper-bounds on performance: count heap references that are still redundant after redundant load elimination.

⇒ Reveals potential room for improvement

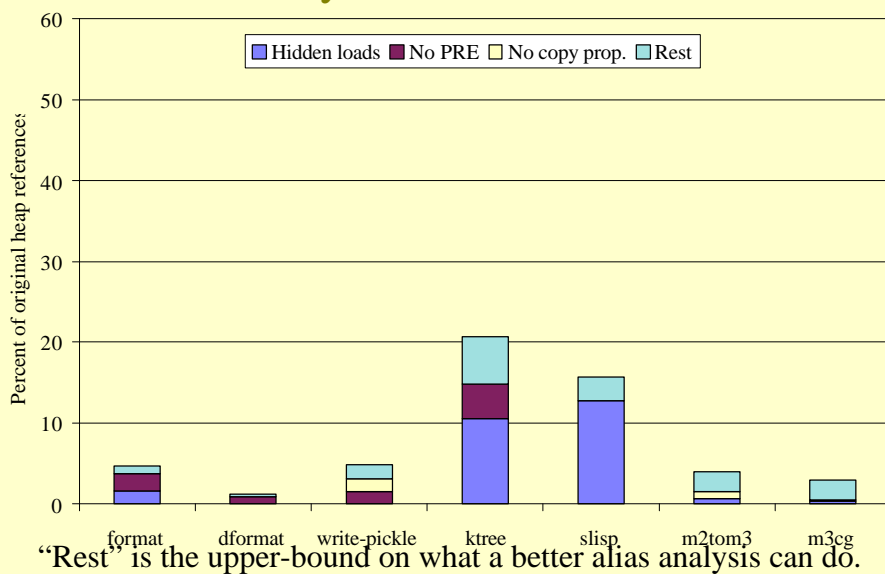
What it does not do:

- Give results for all inputs and optimizations

Limit Evaluation (intraprocedural): Loads that are still redundant



Why still redundant?



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?

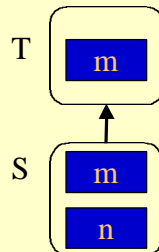
Definitions

- 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

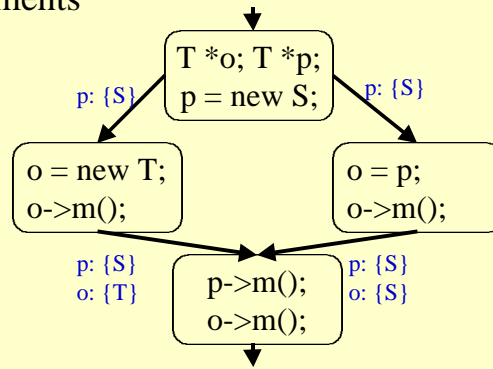


t: T; ...; t->m() can call? **T::m or S::m**

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



Analysis 3: intraprocedural type propagation + tbaa

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

```

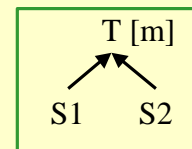
T *s1; S2 **s2;
s2 = ...;
s1 = new S1;
*s2 = new S2;
s1->m();
  
```

Without tbaa

calls S1::m or S2::m

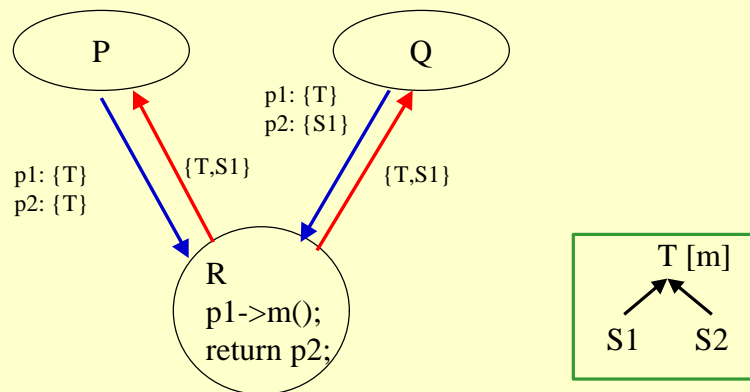
With tbaa

calls S1::m



Analysis 4: interprocedural type propagation

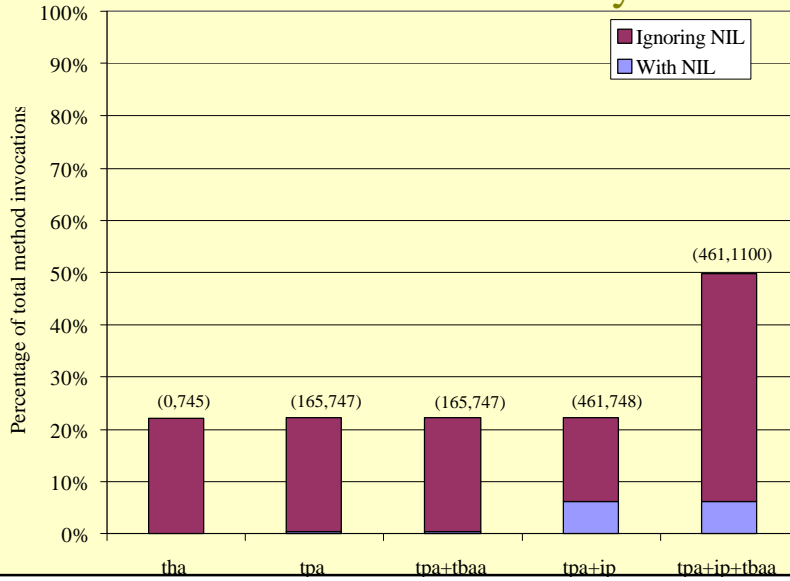
- Similar to intraprocedural type propagation
 - includes monovariant analysis of calls



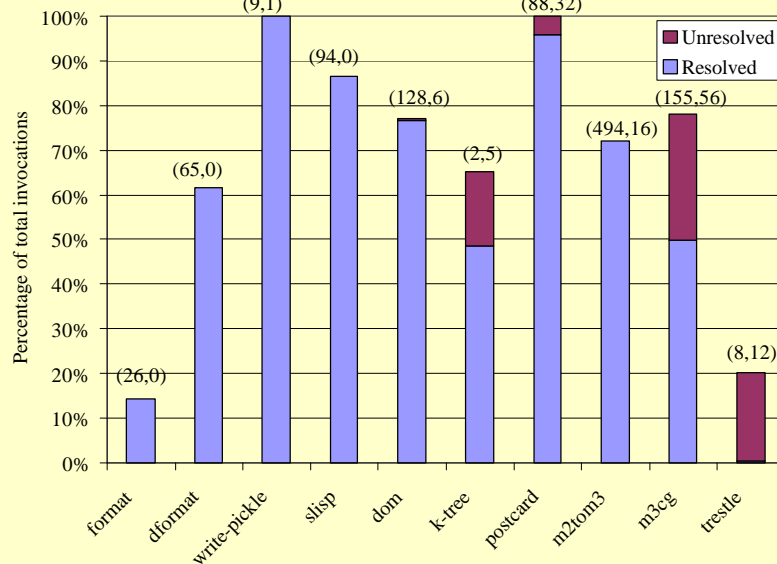
Analysis 5: interprocedural type propagation + tbaa

- Simple extension of interprocedural type propagation with tbaa

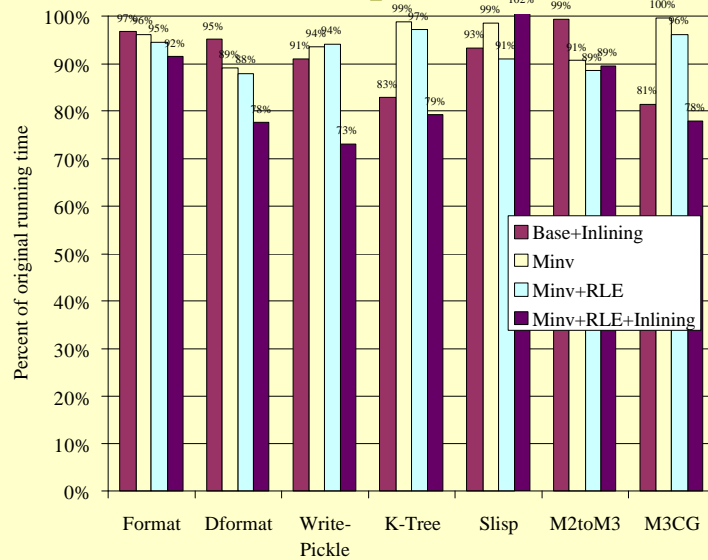
Effectiveness of analyses



How many monomorphic are resolved?



Run-time impact of resolution



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