Type-Based Analysis and Applications

Jens Palsberg

Purdue University

Department of Computer Science

www.cs.purdue.edu/people/palsberg

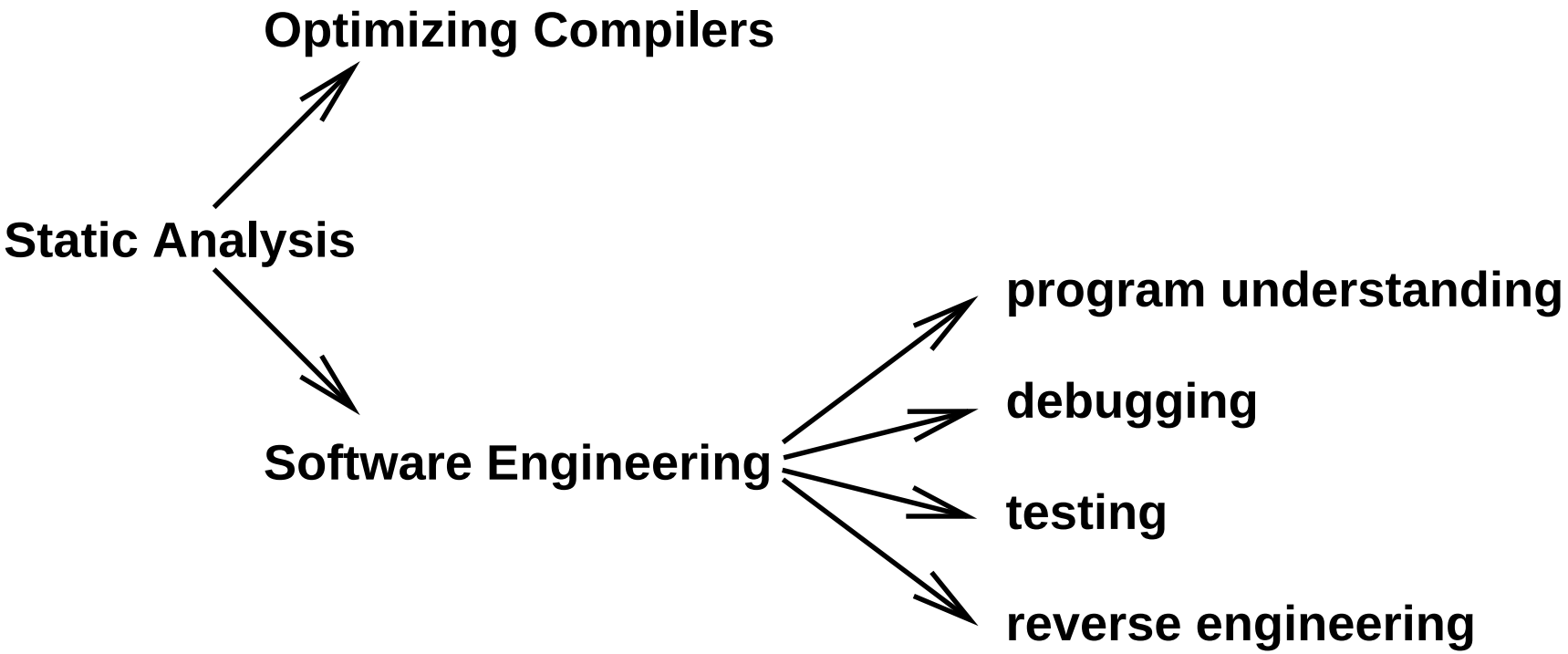
Supported by an NSF CAREER award.

Terminology

A *type-based analysis* assumes that the program type checks, and the analysis takes advantage of that.

- What is an example of a type-based analysis?
- What are the advantages of type-based analysis?
- Is type-based analysis competitive with other approaches to static analysis?
- Which tools use type-based analysis?
- What is the current spectrum of type-based analyses?

Static Analysis: Past Successes



Static Analysis Symposium

International Symposium on Software Testing and Analysis

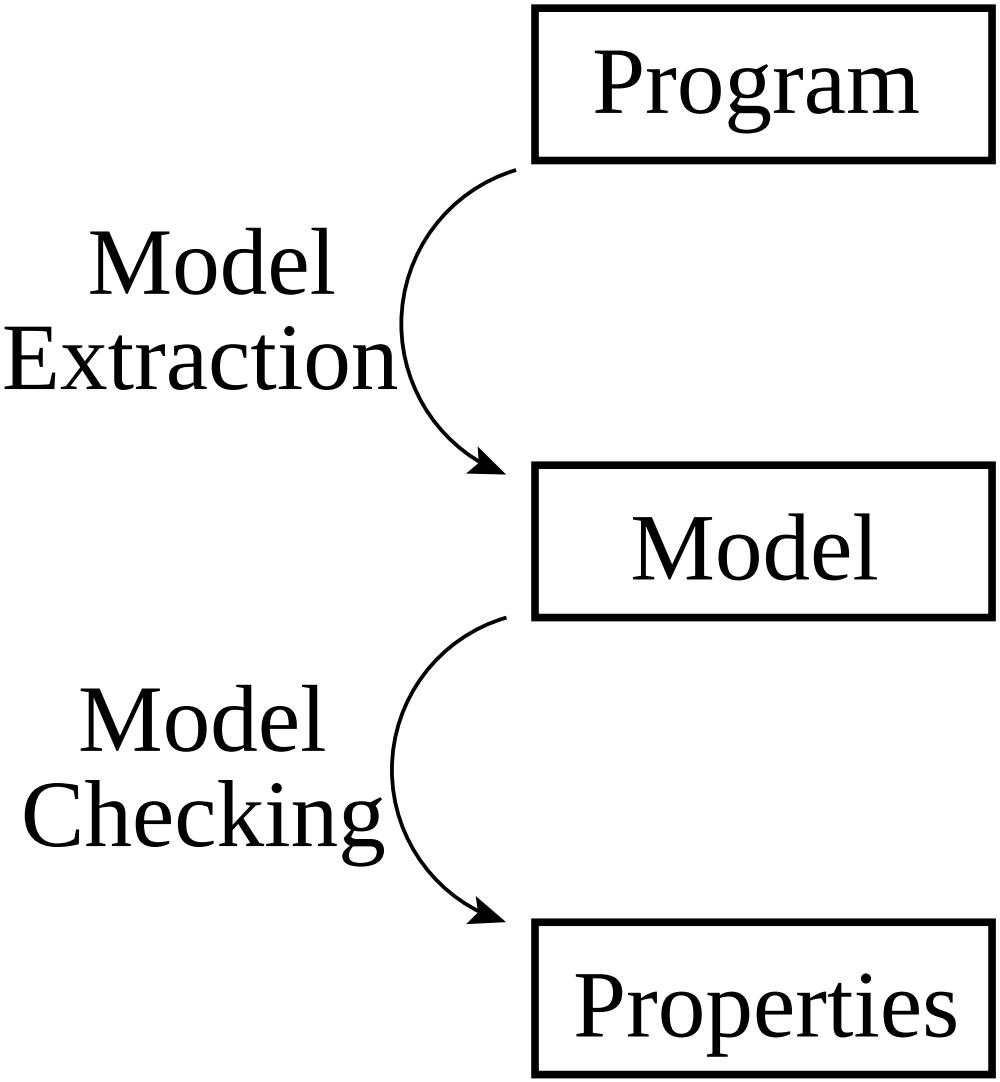
Static Analysis: Future Challenges

Verification of key properties of software:

- real-time properties
- security-related behavior
- power consumption

Highly efficient static analysis for run-time compilation

Scalable static analysis


Program
Model
Extraction
Model
Model
Checking
Properties

The Questions

A *type-based analysis* assumes that the program type checks, and the analysis takes advantage of that.

Can the types help with:

- defining more complicated analyses?
- reasoning about the correctness of an analysis?
- making the static analyses more efficient?

Example: Flow Analysis for the λ-Calculus

Four well-known static analyses:

1. 0-CFA (does not rely on types)
2. type and effect system (type based)
3. sparse flow graph (type based)
4. types as discriminators (type based)

Example: Flow Analysis for the λ-Calculus

$$e \;::=\; x \mid \lambda^l x.e \mid e_1 e_2.$$

What are the possible results of evaluating an expression?

A *flow set* is a set of labels of λ-abstractions.

Goal: compute a flow set that conservatively estimates the possible results.

Conservative = if v is a possible value of e,
then the label of v must be in the flow set of e.

Running example: $F = ((\lambda^1 f.\lambda^2 x.fx)(\lambda^3 a.a))(\lambda^4 b.b)$

$$\begin{aligned} F &\to_\beta (\lambda^2 x.((\lambda^3 a.a)\ x))\ (\lambda^4 b.b) \\ &\to_\beta (\lambda^3 a.a)\ (\lambda^4 b.b) \\ &\to_\beta \lambda^4 b.b. \end{aligned}$$

So, a flow analysis must produce a flow set for F that contains the label 4.

0-CFA

Idea: flow graph.

Nodes $\quad n ::= e \quad$ e occurs in the program E

Edges: [Heintze & McAllester 1997]

$$\lambda^l x.e \to \lambda^l x.e \tag{1}$$

$$\frac{e_1 \to \lambda^l x.e}{x \to e_2} \quad (e_1 e_2 \text{ occurs in } E) \tag{2}$$

$$\frac{e_1 \to \lambda^l x.e}{e_1 e_2 \to e} \quad (e_1 e_2 \text{ occurs in } E) \tag{3}$$

$$\frac{e_1 \to e_2 \quad e_2 \to e_3}{e_1 \to e_3} \tag{4}$$

Idea: the flow set for e is the set of labels of abstractions $\lambda^l x.e'$ such that there is an edge $e \to \lambda^l x.e'$ in the flow graph.

Complexity: $O(n^3)$ time.

Running Example

For F, we can use Rules (1)–(4) to generate the edges:

$$
\begin{aligned}
&f \to \lambda^3 a.a \\
&(\lambda^1 f.\lambda^2 x.fx)(\lambda^3 a.a) \to \lambda^2 x.fx \\
&F \to fx \to a \to x \to \lambda^4 b.b,
\end{aligned}
$$

so by transitivity (Rule (4)), we have $F \to \lambda^4 b.b$,
so the flow set for F is $\{4\}$.

A Simple Type System

Types $\quad t \quad ::= \quad \alpha \mid t \to t \qquad \alpha$ is a type variable

The type rules:

$$A \vdash x : t \qquad (A(x) = t) \tag{5}$$

$$\frac{A[x : s] \vdash e : t}{A \vdash \lambda^l x.e : s \to t} \tag{6}$$

$$\frac{A \vdash e_1 : s \to t \quad A \vdash e_2 : s}{A \vdash e_1 e_2 : t} \tag{7}$$

Running Example

For F, we can use Rules (5)–(7) to construct a type derivation which contains the judgments:

$$\emptyset \vdash \lambda^1 f.\lambda^2 x.fx : ((\alpha \to \alpha) \to (\alpha \to \alpha)) \to ((\alpha \to \alpha) \to (\alpha \to \alpha))$$

$$\emptyset[f : (\alpha \to \alpha) \to (\alpha \to \alpha)] \vdash \lambda^2 x.fx : (\alpha \to \alpha) \to (\alpha \to \alpha)$$

$$\emptyset \vdash \lambda^3 a.a : (\alpha \to \alpha) \to (\alpha \to \alpha)$$

$$\emptyset \vdash \lambda^4 b.b : \alpha \to \alpha$$

$$\emptyset \vdash F : \alpha \to \alpha.$$

A Type and Effect System

Idea: if we have the judgment $A \vdash e : s \xrightarrow{\varphi} t$, then the flow set for e is φ.

Annotated types: $t \quad ::= \quad \alpha \mid t \xrightarrow{\varphi} t$ $\qquad$ φ is a flow set

Revised type rules:

$$A \vdash x : t \qquad (A(x) = t) \tag{8}$$

$$\frac{A[x : s] \vdash e : t}{A \vdash \lambda^{l} x.e : s \xrightarrow{\varphi} t} \quad (l \in \varphi) \tag{9}$$

$$\frac{A \vdash e_1 : s \xrightarrow{\varphi} t \quad A \vdash e_2 : s}{A \vdash e_1 e_2 : t} \tag{10}$$

Running Example

For F, we can use Rules (8)–(10) to construct a type derivation which contains the judgments:

$$\emptyset \vdash \lambda^1 f.\lambda^2 x.fx : ((\alpha \xrightarrow{\{4\}} \alpha) \xrightarrow{\{3\}} (\alpha \xrightarrow{\{4\}} \alpha)) \xrightarrow{\{1\}} ((\alpha \xrightarrow{\{4\}} \alpha) \xrightarrow{\{2\}} (\alpha \xrightarrow{\{4\}} \alpha))$$

$$\emptyset[f : (\alpha \xrightarrow{\{4\}} \alpha) \xrightarrow{\{3\}} (\alpha \xrightarrow{\{4\}} \alpha)] \vdash \lambda^2 x.fx : (\alpha \xrightarrow{\{4\}} \alpha) \xrightarrow{\{2\}} (\alpha \xrightarrow{\{4\}} \alpha)$$

$$\emptyset \vdash \lambda^3 a.a : (\alpha \xrightarrow{\{4\}} \alpha) \xrightarrow{\{3\}} (\alpha \xrightarrow{\{4\}} \alpha)$$

$$\emptyset \vdash \lambda^4 b.b : \alpha \xrightarrow{\{4\}} \alpha$$

$$\emptyset \vdash F : \alpha \xrightarrow{\{4\}} \alpha,$$

so the flow set for F is $\{4\}$.

Sparse Flow Graphs

Idea: *sparse* flow graph; no transitive closure [Heintze & McAllester 97].

Nodes $\quad n ::= e \mid \mathrm{dom}(n) \mid \mathrm{ran}(n) \qquad e$ occurs in the program E

Edges:

$$x \to \mathrm{dom}(\lambda^l x.e) \qquad (\lambda^l x.e \text{ occurs in } E) \tag{11}$$

$$\mathrm{ran}(\lambda^l x.e) \to e \qquad (\lambda^l x.e \text{ occurs in } E) \tag{12}$$

$$e_1 e_2 \to \mathrm{ran}(e_1) \qquad (e_1 e_2 \text{ occurs in } E) \tag{13}$$

$$\mathrm{dom}(e_1) \to e_2 \qquad (e_1 e_2 \text{ occurs in } E) \tag{14}$$

$$\frac{n_1 \to n_2 \quad n \to \mathrm{ran}(n_1)}{\mathrm{ran}(n_1) \to \mathrm{ran}(n_2)} \tag{15}$$

$$\frac{n_1 \to n_2 \quad n \to \mathrm{dom}(n_2)}{\mathrm{dom}(n_2) \to \mathrm{dom}(n_1)} \tag{16}$$

Running Example

For F, we can use Rules (11)–(16) to generate the edges:

$$
\begin{aligned}
& f \to \mathrm{dom}(\lambda^1 f.\lambda^2 x.fx) \to \lambda^3 a.a \\
& (\lambda^1 f.\lambda^2 x.fx)(\lambda^3 a.a) \to \mathrm{ran}(\lambda^1 f.\lambda^2 x.fx) \to \lambda^2 x.fx \\
& F \to \mathrm{ran}((\lambda^1 f.\lambda^2 x.fx)(\lambda^3 a.a)) \to \mathrm{ran}(\mathrm{ran}(\lambda^1 f.\lambda^2 x.fx)) \\
& \quad \to \mathrm{ran}(\lambda^2 x.fx) \to fx \to \mathrm{ran}(f) \\
& \quad \to \mathrm{ran}(\mathrm{dom}(\lambda^1 f.\lambda^2 x.fx)) \to \mathrm{ran}(\lambda^3 a.a) \to a \\
& \quad \to \mathrm{dom}(\lambda^3 a.a) \to \mathrm{dom}(\mathrm{dom}(\lambda^1 f.\lambda^2 x.fx)) \to \mathrm{dom}(f) \\
& \quad \to x \to \mathrm{dom}(\lambda^2 x.fx) \to \mathrm{dom}(\mathrm{ran}(\lambda^1 f.\lambda^2 x.fx)) \\
& \quad \to \mathrm{dom}((\lambda^1 f.\lambda^2 x.fx)(\lambda^3 a.a)) \\
& \quad \to \lambda^4 b.b,
\end{aligned}
$$

so the flow set for F is $\{4\}$.

Simply Typed $\implies$ finite and sparse graph, and same result as 0-CFA.

Simple Small Types $\implies$ $O(n^2)$ time.

Types as Discriminators

Flow set for e = which abstractions in E have the same type as e ?

For the running example F, we have:

$$
\begin{aligned}
&\emptyset \vdash \lambda^1 f.\lambda^2 x.fx : ((\alpha \to \alpha) \to (\alpha \to \alpha)) \to \\
&\qquad\qquad ((\alpha \to \alpha) \to (\alpha \to \alpha)) \\
&\emptyset[f : (\alpha \to \alpha) \to (\alpha \to \alpha)] \vdash \lambda^2 x.fx : \\
&\qquad\qquad (\alpha \to \alpha) \to (\alpha \to \alpha) \\
&\emptyset \vdash \lambda^3 a.a : (\alpha \to \alpha) \to (\alpha \to \alpha) \\
&\emptyset \vdash \lambda^4 b.b : \alpha \to \alpha \\
&\emptyset \vdash F : \alpha \to \alpha.
\end{aligned}
$$

There is exactly one abstraction in F with type $\alpha \to \alpha$, namely $\lambda^4 b.b$, so the flow set for F is $\{4\}$.

Advantages of type-based analysis

Simplicity

$$\frac{A[x:s] \vdash e : t}{A \vdash \lambda^l x.e : s \xrightarrow{\varphi} t} \quad (l \in \varphi)$$

Efficiency

Types can make almost anything go faster!

Correctness

Type soundness: well-typed programs cannot go wrong [Milner 78].

Similar story for type and effect systems.

Competitiveness

Main approaches to static analysis:

data flow analysis, constraint-based analysis, abstract interpretation, …

Important similarities!

Correctness	Algorithm
type rules	constraints
type and effect system	constraints

Define abstract domains in terms of types.

The types are a *lingua franca* when comparing analyses.

Tools that use type-based analysis

Work on programs written in C++, Java, Modula 3, and Standard ML.

Method Inlining

Object-oriented virtual call site $e.m(\ldots)$ inline?

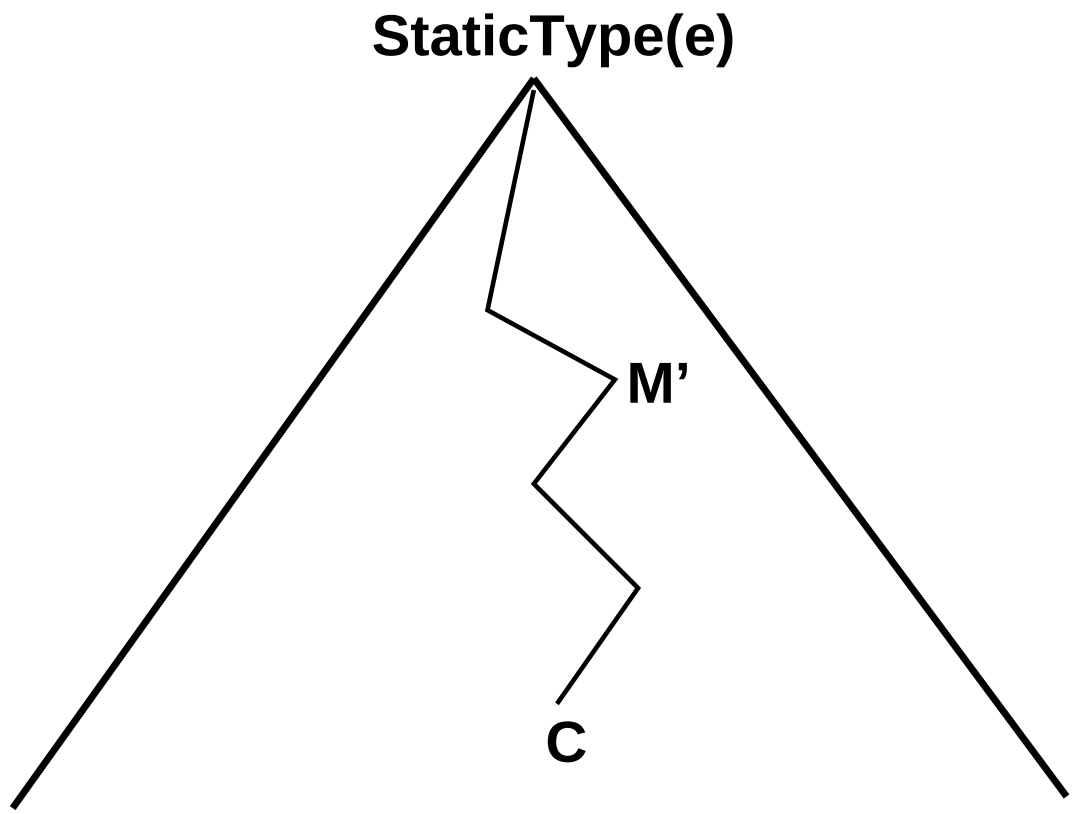
Approach: types as discriminators, taking advantage of subtyping.

StaticType(e)	the static type of the expression e,
SubTypes(t)	the set of declared subtypes of type t
StaticLookup(C,m)	definition (if any) of a method with name m that one finds when starting a static method lookup in the class C.

Method Inlining

Class Hierarchy Analysis (CHA) [Dean, Grove, & Chambers 1995]

For the virtual call site $e.m(\ldots)$, and
each class $C \in \mathit{SubTypes}(\mathit{StaticType}(e))$
where $\mathit{StaticLookup}(C, m) = M'$,
CHA determines that M' is a method that can be invoked.



Method Inlining

Rapid Type Analysis (RTA) [Bacon & Sweeney 1996]
takes class-instantiation information into account.

Idea: first collect the set S of all classes C for which there is an occurrence of "new C()" in the program.

For the virtual call site $e.m(\ldots)$, and
each class $C \in \mathit{SubTypes}(\mathit{StaticType}(e))$
where $\mathit{StaticLookup}(C,m) = M'$ and $C \in S$,
RTA determines that M' is a method that can be invoked.

Next idea: associate a single distinct set (like S) with each class, method, and/or field in an application [Tip & Palsberg 2000].

If one associates a set with each expression, then the result is 0-CFA [Palsberg & Schwartzbach 1991].

Method Inlining

Idea: first use CHA/RTA to determine a call graph approximation,
then use a 0-CFA-like technique to propagate class information
[Sundaresan et al. 2000]:

Observation: CHA, RTA, and others are whole-program analyses.
What about libraries?

Idea: specify what to expect from the library [Sweeney & Tip 2000].

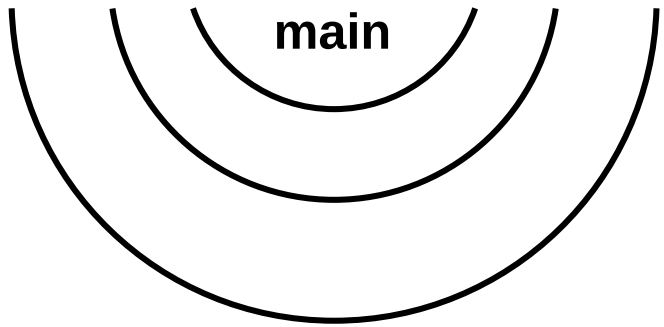
The Swift compiler for Java [Ghemawat, Randall, & Scales 2000]:

frontend:	Java $\rightarrow$ intermediate representation with annotated types
backend:	uses the annotations for method inlining, etc.

Application Extraction

Which methods that are reachable from the main method?

Idea: flow analysis + reachability analysis



CHA + reachability analysis:

uses a single set variable R (for "reachable methods")
that ranges over sets of methods

Constraints:

1. $main \in R$ ($main$ denotes the main method)

2. For each method M, each virtual call site $e.m(\ldots)$ occurring in M, and each class $C \in \mathit{SubTypes}(\mathit{StaticType}(e))$
where $\mathit{StaticLookup}(C,m) = M'$:
$$(M \in R) \Rightarrow (M' \in R).$$

Application Extraction

RTA + reachability analysis:

uses both a set variable R ranging over sets of methods, and
a set variable S which ranges over sets of class names.

Tool for Java: Jax [Tip et al.]

Constraints:

1. $main \in R$ ($main$ denotes the main method)

2. For each method M, each virtual call site $e.m(\ldots)$ occurring in M, and each class $C \in \mathit{SubTypes}(\mathit{StaticType}(e))$
   where $\mathit{StaticLookup}(C,m) = M'$:
   $$(M \in R) \wedge (C \in S) \;\Rightarrow\; (M' \in R).$$

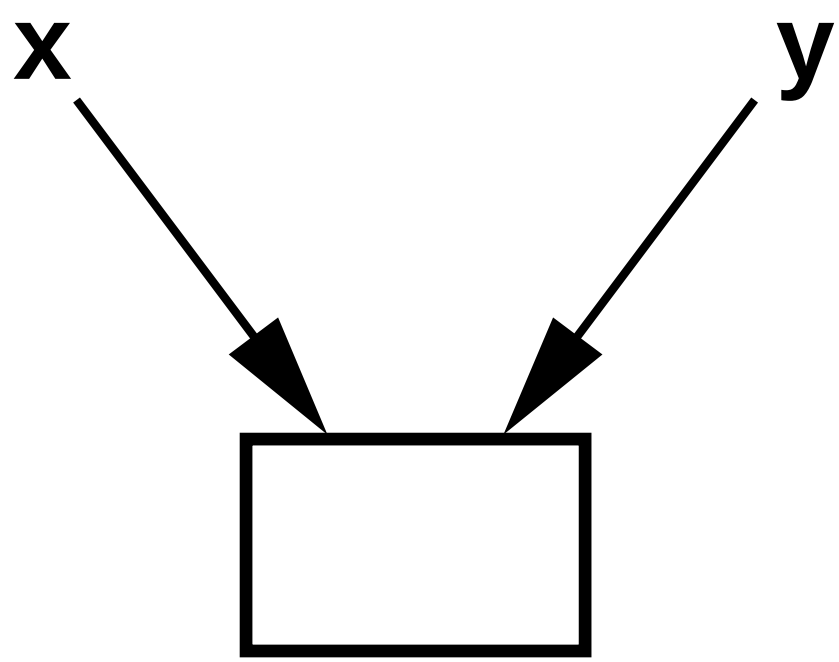
3. For each method M, and for each "new C()" occurring in M:
   $$(M \in R) \;\Rightarrow\; (C \in S).$$

Redundant-Load Elimination

Combines loop-invariant code motion and
common-subexpression elimination.

Reorders statements that may do pointer accesses $\Longrightarrow$
can benefit from alias information.

Two access paths are said to be possibles aliases if
they may refer to the same variable.

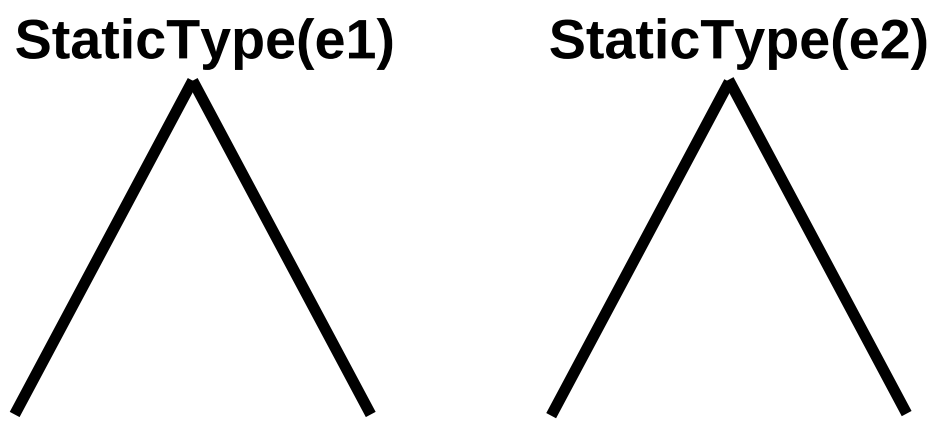


Redundant-Load Elimination

Three type-based alias analyses [Diwan, McKinley, & Moss 1998]
use types as discriminators.

TypeDecl: two expressions e_1 and e_2 cannot be aliases if

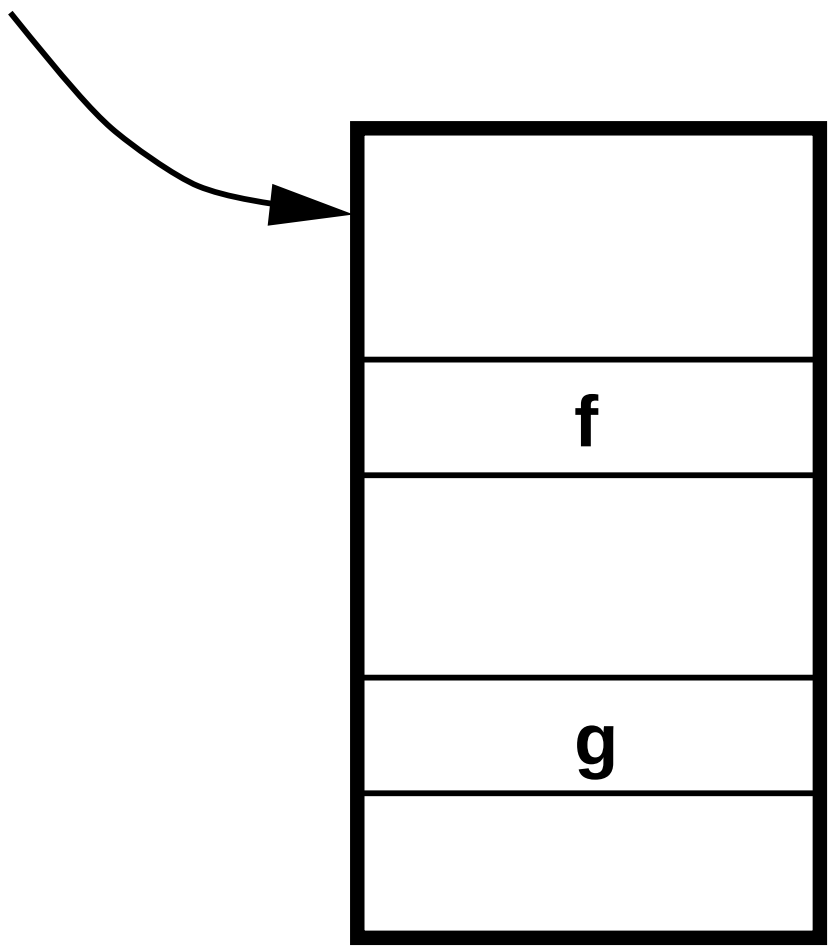
$$\mathit{SubTypes}(\mathit{StaticType}(e_1)) \cap \mathit{SubTypes}(\mathit{StaticType}(e_2)) = \emptyset.$$



Redundant-Load Elimination

FieldTypeDecl: TypeDecl + things like:

two expressions $e_1.f$ and $e_2.g$ cannot be aliases if $f \neq g$.



Redundant-Load Elimination

SMTypeRefs: includes a type-based flow analysis.

Idea: two expressions e_1 and e_2 cannot be aliases if
the program never assigns an object of type *StaticType*(e_1)
to a reference of type *StaticType*(e_2), or vice versa.

Flow graph:

Nodes $\quad n ::= C \quad$ C is the name of a class in the program

Edges:

$x = e$	$x : A \quad e : B$	$A \leftarrow B$
$y = \text{new } C()$	$y : B$	$B \leftarrow \boxed{\text{C}}$

Redundant-Load Elimination

Experiments: FieldTypeDecl and SMTypeRefs are good;
TypeDecl seems to be too imprecise.

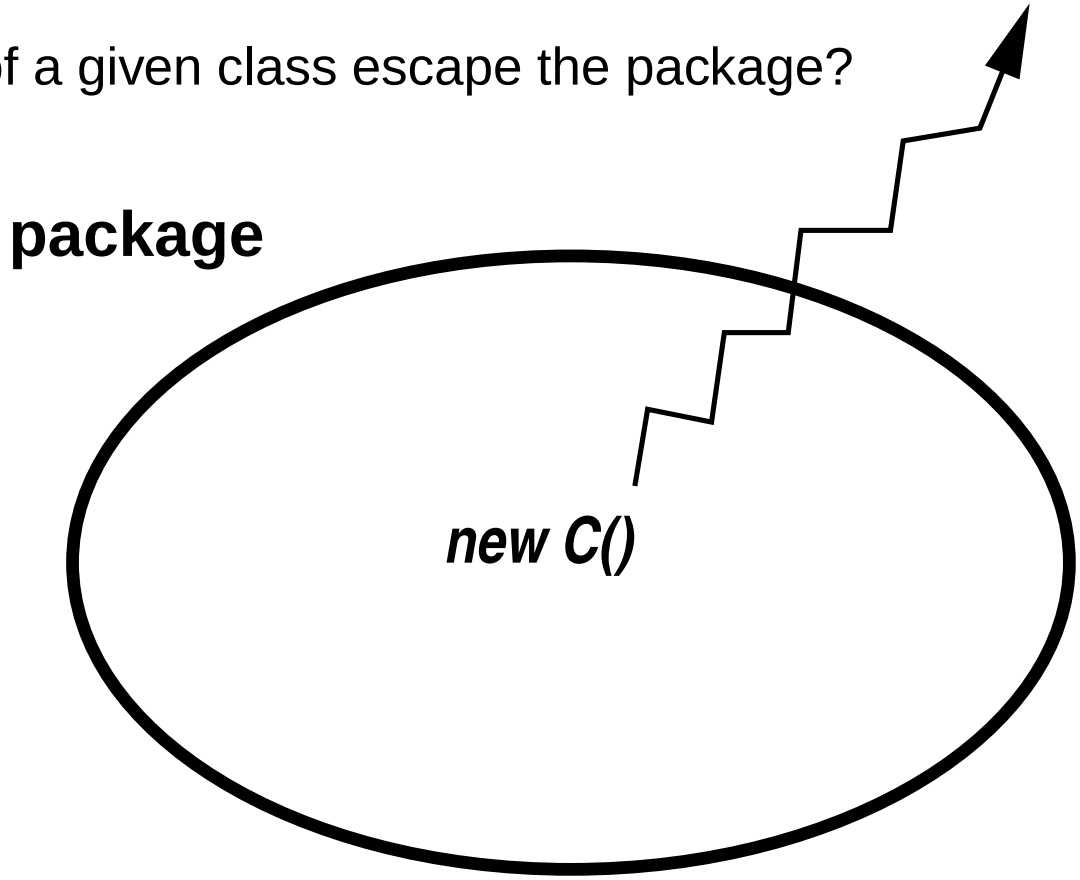
[Fink, Knobe, & Sarkar 2000] used a flow-sensitive version of FieldTypeDecl in their implementation of redundant-load and dead-store elimination.

[Hosking, Nystrom, Whitlock, Cutts, & Diwan 2001]:

- partial-redundancy elimination
- uses FieldTypeDecl
- experience: mixed

Encapsulation Checking (Escape Analysis)

Can objects of a given class escape the package?



A class whose objects do not escape is *confined* [Bokowski & Vitek 1999]

The objects a confined class are *encapsulated* in the package.

Type-based analysis for identifying confined classes in Java bytecode

uses constraints + SMTypeRefs [Grothoff, Vitek, & Palsberg 2001]

Race Detection

Multi-threaded

Race condition: two threads manipulate a shared data structure simultaneously, without synchronization.

Use locks

Type-based analysis that detects race conditions in Java programs

- type and effect system + tool [Flanagan & Freund 2000]
- requires adding some type annotations to the Java code

Memory Management

Store = stack of regions [Tofte & Talpin 1997]

for call-by-value functional languages

Region inference determines where regions can be allocated/deallocated

type and effect system

Implementation for Standard ML [Birkedal, Tofte, & Vejlstrup 1996]

compares well with garbage collection:

can save space and compete on speed

Other type-based analyses: type and effect systems

side-effect analysis	binding-time analysis
strictness analysis	totality analysis
callability analysis	flow analysis
trust analysis	secure information flow analysis
closure conversion	resource allocation in compilers
continuation allocation	dependency analysis
communication analysis	elimination of useless variables

and more!

– many of them have been proved correct
– most have not yet been implemented for a full-fledged language
– some have been implemented for a toy language
– some still need an algorithm for performing the analysis.

A survey of the methodology behind type and effect systems:
[Nielson & Nielson 1999]

Conclusion

Most of the surveyed tools use types as discriminators.

Most of the theoretical studies use type and effect systems.

Ideal: both proof of correctness and convincing experimental results.

Type-based analysis is a promising approach to achieving both with a reasonable effort.

Further information about type-based analysis and links to many of the cited papers are available from:

http://www.cs.purdue.edu/homes/palsberg/tba/

Secure Software Systems Group

Purdue University

http://www.cs.purdue.edu/s3/

Faculty: Antony Hosking, Jens Palsberg, Jan Vitek.

16 Ph.D. students, 2 M.S. students, 3 undergraduate students.

Sample projects: Java security, bytecode compression, interoperability of software systems, real-time system verification, software watermarking, high-performance persistent object storage.

Funding: NSF, DARPA, CERIAS, Sun Microsystems, Lockheed Martin, IBM, Motorola, and Intel.