Supported by an NSF CAREER award.

www.cs.purdue.edu/people/palsberg

Department of Computer Science

Purdue University

Jens Palsberg

Type-Based Analysis and Applications
What is type-based analysis?

Which tools use type-based analysis?

Is type-based analysis competitive with other approaches to static analysis?

What are the advantages of type-based analysis?

What is an example of a type-based analysis?

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

Reverse Engineering
Software Engineering
Testing
Debugging
Program Understanding
Optimizing Compilers

Past Successes
Static Analysis: Future Challenges

Verification of key properties of software:

- Real-time properties
- Security-related behavior
- Power consumption
- Scalable static analysis

Highly efficient static analysis for run-time compilation

Verification of key properties of software: Scalable static analysis
The Questions

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

• Making the static analyses more efficient?

• Reasoning about the correctness of an analysis?

• Defining more complicated analyses?

Can the types help with:
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)
So, a flow analysis must produce a flow set for \( F \) that contains the label 4.

\[
x \quad \lambda \quad e \quad e_1 \quad e_2
\]

What are the possible results of evaluating an expression?

\[
x \quad \lambda \quad e \quad e_1 \quad e_2 \quad F_1 \quad \text{Example: Flow Analysis for the \( \lambda \)-Calculus}
\]
Complexity: $O(3^n)$ time.

There is an edge $e \leftarrow \lambda x_1 y_1 \in$ in the flow graph.

Idea: the flow set for $e$ is the set of labels of abstractions $\lambda x_1 y_1$ such that $e_1 e_2$ occurs in $E$.

\[
\begin{aligned}
\text{(4)} & \quad e_3 \leftarrow e_1 e_2 \\
\text{(3)} & \quad \text{($e_1 e_2$ occurs in $E$)} \quad \quad e_3 \leftarrow e_1 e_2 \\
\text{(2)} & \quad \text{($e_1 e_2$ occurs in $E$)} \quad \quad x \leftarrow e_2 \\
\text{(1)} & \quad \lambda x_1 y_1 \leftarrow e_1 e_2 x_1 y_1
\end{aligned}
\]

Edges: [Heintze & McAllester 1997]

Nodes: Flow graph.

Nodes $e = :: u$ occur in the program $E$. 

Idea: Flow graph.
For $F$, we can use Rules (1)–(4) to generate the edges:

{4}

so the flow set for $F$ is

$q.q.\gamma \leftarrow x \leftarrow a \leftarrow x.f \leftarrow F$

$xf.x.\gamma \leftarrow (a.a.\gamma)(xf.x.\gamma.f.\gamma)$

$a.a.\gamma \leftarrow f$

So by transitivity (Rule (4)), we have $F$ is $q.q.\gamma$.

Running Example
(7) \[ \frac{1 : \forall \alpha \in \mathbb{E}}{s : \forall \alpha \in \mathbb{E} \vdash A \quad t \leftarrow s : 1 : \forall \alpha \in \mathbb{E} \vdash A} \]

(8) \[ \frac{1 \leftarrow s : \forall \alpha \cdot x \vdash A \quad \forall \lambda \lambda x}{1 : \forall \alpha \vdash [s : x] \forall} \]

(9) \[ (i = (x)A) \quad 1 : x \vdash A \]

The type rules:

- A is a type variable
- \( 1 \leftarrow 1 \mid a = : 1 \)

A Simple Type System
Running Example

For $F$, we can use Rules (g)–(i) to construct a type derivation which contains the judgements:

\[
\begin{align*}
\cdot \alpha & \leftarrow \alpha : \forall \rightarrow \emptyset \\
\alpha & \leftarrow \alpha : \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall
\end{align*}
\]
ATypeandEffectSystem

Idea: if we have the judgment $A$, then the flow set for $\sigma$ is $\phi$.  

Revised type rules:

Annotated types: if we have the judgment $A \vdash x : \sigma$, then the flow set for $\sigma$ is $\phi$.  

A Type and Effect System
Running Example

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

\[
\begin{align*}
\{4\} &\quad F \vdash \emptyset \\
\alpha &\quad \{4\} \\
\alpha &\quad \{4\} \\
\lambda f &\quad \{4\} \\
\lambda x &\quad \{4\} \\
\lambda x &\quad \{4\} \\
\lambda 0 &\quad \{4\} \\
\lambda 0 &\quad \{4\} \\
(\alpha \vdash \alpha) &\quad \{3\} \\
(\alpha \vdash \alpha) &\quad \{3\} \\
(\alpha \vdash \alpha) &\quad \{3\} \\
(\alpha \vdash \alpha) &\quad \{3\} \\
x f \cdot x \cdot \gamma &\vdash \left[(\alpha \vdash \alpha) \vdash \{3\} (\alpha \vdash \alpha) : f] \emptysetight. \\
((\alpha \vdash \alpha) \vdash \{3\} (\alpha \vdash \alpha)) &\vdash x f \cdot x \cdot \gamma \cdot f \cdot \gamma \vdash \emptyset
\end{align*}
\]

\{4\} is the flow set for \( F \).
Sparse Flow Graphs

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

Nodes

- $n_1 \leadsto (\exists u)(\forall e \in \mathcal{E}) \quad \exists e \in \mathcal{E}$
- $n_2 \leadsto (\exists u)(\forall e \in \mathcal{E}) \quad \exists e \in \mathcal{E}$
- $n_1 \leadsto (\forall e \in \mathcal{E}) \quad \forall e \in \mathcal{E}$
- $n_2 \leadsto (\forall e \in \mathcal{E}) \quad \forall e \in \mathcal{E}$

Edges:

- $(\exists u)(\forall e \in \mathcal{E}) \leadsto (\forall e \in \mathcal{E})$
Running Example

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

$$
\begin{align*}
\text{So the flow set for } F \text{ is } & \{ 4 \} . \\
\begin{array}{l}
q :: q, 4 \vdash \gamma \\
((p \cdot p \cdot \gamma')(xf \cdot x \cdot \gamma \cdot f \cdot x \cdot \gamma)) \text{ dom} & \leftarrow \\
((xf \cdot x \cdot \gamma \cdot f \cdot x \cdot \gamma) \text{ ran} \text{ dom} & \leftarrow (xf \cdot x \cdot \gamma) \text{ dom} & \leftarrow x \\
(f) \text{ dom} & \leftarrow ((xf \cdot x \cdot \gamma \cdot f \cdot x \cdot \gamma) \text{ dom} & \leftarrow (p \cdot p \cdot \gamma) \text{ dom} & \leftarrow \\
p & \leftarrow (p \cdot p \cdot \gamma) \text{ ran} & \leftarrow ((xf \cdot x \cdot \gamma \cdot f \cdot x \cdot \gamma) \text{ ran} & \leftarrow \\
(f) \text{ ran} & \leftarrow xf & \leftarrow (xf \cdot x \cdot \gamma) \text{ ran} & \leftarrow \\
((xf \cdot x \cdot \gamma \cdot f \cdot x \cdot \gamma) \text{ ran} & \leftarrow ((p \cdot p \cdot \gamma')(xf \cdot x \cdot \gamma \cdot f \cdot x \cdot \gamma)) \text{ ran} & \leftarrow F \\
xf \cdot x \cdot \gamma & \leftarrow (xf \cdot x \cdot \gamma \cdot f \cdot x \cdot \gamma) \text{ ran} & \leftarrow (p \cdot p \cdot \gamma'(xf \cdot x \cdot \gamma \cdot f \cdot x \cdot \gamma) \\
p \cdot p \cdot \gamma & \leftarrow (xf \cdot x \cdot \gamma \cdot f \cdot x \cdot \gamma) \text{ dom} & \leftarrow f
\end{array}
\end{align*}
$$

Simply Typed Small Types

Finite and Sparse Graph, and same result as 0-CFA.
Types as Discriminators

Flowset for \( F \) is

\[ \begin{align*}
\{ \{ 3 \} \} \quad \text{so the flow set for } F \text{ is } \{ 4 \} .
\end{align*} \]

There is exactly one abstraction in \( F \) with type \( \alpha \), namely \( \lambda q. q \cdot \alpha \),

\[
\begin{align*}
\alpha & \leftarrow \alpha : F \rightarrow 0 \\
\alpha & \leftarrow \alpha : q \cdot q \cdot \chi \rightarrow 0 \\
(\alpha \leftarrow \alpha) & \leftarrow (\alpha \leftarrow \alpha) : \forall \alpha \forall \chi \chi \rightarrow 0 \\
(\alpha \leftarrow \alpha) & \leftarrow (\alpha \leftarrow \alpha) \\
: x f \cdot x \cdot \chi & \leftarrow [(\alpha \leftarrow \alpha) \leftarrow (\alpha \leftarrow \alpha) : f] 0 \\
((\alpha \leftarrow \alpha) & \leftarrow (\alpha \leftarrow \alpha)) \\
\leftarrow ((\alpha \leftarrow \alpha) \leftarrow (\alpha \leftarrow \alpha)) : x f \cdot x \cdot \chi \cdot f \cdot \chi \rightarrow 0 \\
\end{align*} \]

For the running example \( F \), we have:

\[ F = \{ \alpha \} \]

Flow set for \( \alpha \) = which abstractions in \( F \) have the same type as \( \alpha \)?

\underline{Types as Discriminators}
Advantages of type-based analyses

Similiar story for type and effect systems.

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

Correctness

Types can make almost anything go faster!

Efficiency

Simplicity

\[
\phi \vdash t \\
\frac{1 \vdash s : \forall x \forall \gamma \forall \forall}{1 \vdash \phi 
\frac{1 \vdash s : \forall x \forall \gamma \forall \forall}{1 \vdash x : \forall \forall \forall}
\]
The types are a lingua franca when comparing analyses.

Define abstract domains in terms of types.

<table>
<thead>
<tr>
<th>Type and effect system constraints</th>
<th>Type rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td>Correctness</td>
</tr>
</tbody>
</table>

Important similarities!

Main approaches to static analyses:
- Data flow analysis
- Constraint-based analysis
- Abstract interpretation
- Abstract interpretation

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

Tools that use type-based analyses
Approach: types as discriminators, taking advantage of subtyping.

in the class C.

that one finds when starting a static method lookup

definition (if any) of a method with name m

the set of declared subtypes of type t

the static type of the expression e.

\[ \text{StaticLookup}(C, m) \]

\[ \text{SubTypes}(t) \]

\[ \text{StaticType}(e) \]

Object-oriented virtual call site

\( \text{Method Inlining} \)

\( \text{inline?} \)

\( e. m(\cdots) \)
Class Hierarchy Analysis (CHA) [Dean, Grove, & Chambers 1995]

Method Inlining

For the virtual call site e.m(···), and each class C ∈ SubTypes(StaticType(e))

where StaticLookup(C, m) = m', and CHA determines that M' is a method that can be invoked.

static(e)
Method Inlining

Rapid Type Analysis (RTA) [Bacon & Sweeney 1996]

If one associates a set with each expression, then the result is 0-CFA

and/or field in an application [Tip & Palserberg 2000].

Next idea: associate a single distinct set (like $S$) with each class, method, and/or field in an application.

RTA determines that $M$, is a method that can be invoked.

where StaticLookup($C$, $m$) = $M$, and $C \in S$,

each class $C \in$ SubTypes(StaticType($e$)) and

For the virtual call site $e.m$, and $C \in S$,

Ideas: First collect the set $S$ of all classes $C$ for which there is an occurrence

of "new $C()" in the program.

takes class-instantiation information into account.

Method Inlining
Method Inlining

Idea: First use CHA/RTA to determine a call graph approximation, then use a 0-CFA-like technique to propagate class information.

Observation: CHA, RTA, and others are whole-program analyses.

What about libraries?

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

Observation: CHA, RTA, and others are whole-program analyses.

[Sun et al. 2000]:
then use a 0-CFA-like technique to propagate class information.

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

frontend: Java
intermediate representation with annotated types
backend: use the annotations for method Inlining, etc.

Idea: specify what to expect from the library [Sweeney & Tip 2000].
Which methods that are reachable from the main method?

Idea: Flow analyses + reachability analyses

Constraints:

1. \( \text{main} \in R \) (\( \text{main} \) denotes the main method)

2. For each method \( M \), each virtual call site \( e \cdot m \) occurring in \( M \), and each class \( C \in \text{SubTypes}(\text{StaticType}(e)) \):

\[
\text{StaticLookup}(C \cdot m) = M \iff M \in R
\]

CHA + Reachability analyses:

Uses a single set variable \( R \) (for "reachable methods")

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. $\text{main} \in R$ (denotes the main method)
2. For each method $M$, each virtual call site $e.m(...)$ occurring in $M$, and each class $C \in \text{SubTypes}(\text{StaticType}(e))$, where $\text{StaticLookup}(C,m) = M'$:
   
   \[
   (M' \in R) \land (C \in S) \Rightarrow (M' \in R).
   \]
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.

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

They can benefit from alias information.

Common-subexpression elimination

Combines loop-invariant code motion and
Redundant-Load Elimination

Type-based alias analysis [Diwan, McKinley, & Moss 1998]

\[ \emptyset = \text{SubTypes}(\text{StaticType}(e_1)) \cup \text{SubTypes}(\text{StaticType}(e_2)) \]

Typedef: two expressions \( e_1 \) and \( e_2 \) cannot be aliases if

use types as discriminators.

Three type-based alias analyses [Diwan, McKinley, & Moss 1998]
Redundant-Load Elimination

FieldTypeDecl: TypeDecl + things like:

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

```
  f  
```

```
  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:*

```
\[ \text{Flow graph:} \]
```

```
\[
\begin{array}{c}
\text{Nodes:} \\
C \text{ is the name of a class in the program} \\
\end{array}
\]
```

```
\[
\begin{array}{c}
\text{Edges:} \\
C \leftarrow B \quad B \leftarrow A \quad B \leftarrow e \quad e \leftarrow x \\
\text{new } C() \quad e = x \\
\end{array}
\]
```

Redundant-Load Elimination

Experiments: mixed

uses FieldTypedecl

partial-redundancy elimination

[Hosking, Nyström, Whitleck, Cutts, & Diwan 2001]:

partially redundant elimination of redundant-load and dead-store elimination.

[Fink, Knobe, & Sarker 2000] used a flow-sensitive version of FieldTypedecl:

TypeDecl seems to be too imprecise.

Experiments: FieldTypedecl and SMT interpreters are good.

Redundant-Load Elimination
Encapsulation Checking (Escape Analysis)

Can objects of a given class escape the package?

The objects of a confined class are encapsulated in the package. A class whose objects do not escape is confined [Bokowski & Vitek 1999].

Type-based analysis for identifying confined classes in Java bytecode uses constraints + SMT/SMT solvers [Czajkowski, Vitek, & Paldberg 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

Type and effect system + tool [Flanagan & Freund 2000]

Use locks
Memory Management

store = stack of regions

for call-by-value functional languages

Tofte & Talpin 1997]

region inference determines where regions can be allocated/deallocated

type and effect system

implementation for Standard ML [Birkedal, Tofte, & Veijlstrup 1996]

compares well with garbage collection:

Save space and complete on speed

of store

Tofte & Talpin 1997]
A survey of the methodology behind type and effect systems:

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

Other type-based analyses: type and effect systems

- Strictness analyses
- Callability analyses
- Trust analyses
- Continuation allocation
- Closure conversion
- Dependency analyses
- Resource allocation in compilers
- Secure information flow analyses
- Binding-time analyses
- Elimination of useless variables
- Communication analysis
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/
Faculty: Antony Hosking, Jens Palsberg, Jan Vitek.

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.

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

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

Purdue University

Secure Software Systems Group