This TEXT is normal. 1 + 2 = \vec{v} • • •
This TEXT is red. 1 + 2 = \vec{v} • • •
This TEXT is green. 1 + 2 = \vec{v} • • •
This TEXT is blue. 1 + 2 = \vec{v} • • •
This TEXT is yellow. 1 + 2 = \vec{v} • • •
This TEXT is darkred. 1 + 2 = \vec{v} • • •
This TEXT is darkgreen. 1 + 2 = \vec{v} • • •
This TEXT is darkblue. 1 + 2 = \vec{v} • • •
This TEXT is darkyellow. 1 + 2 = \vec{v} • • •
This TEXT is purple. 1 + 2 = \vec{v} • • •

This TEXT is grey. 1 + 2 = \vec{v} • • •
This TEXT is black. 1 + 2 = \vec{v} • • •
This TEXT is normal. 1 + 2 = \vec{v} • • •
Foreword

These are slides presented by Steffen Jost at ESOP/ETAPS’06 on Monday, 27 March 2006, Vienna, Austria
(Yellow slides were not shown but added later...)

I recommend interested people to read our ESOP’06 paper:
  Type-based amortised heap-space analysis
  (for an object-oriented language)

Further information can be found at my homepage
  http://www.dcs.st-and.ac.uk/~jost

Feel free to contact me via email: jost@dcs.st-andrews.ac.uk
Type-based amortised analysis

Martin Hofmann and Steffen Jost
LMU Munich (Bavaria) / St Andrews (Scotland)

Vienna, 27 March 2006
The Idea:

**Amortised Analysis**
Well-known technique used for Complexity Theory analysis

**Linear Programming**
Well-known efficient technique of solving linear constraints

**Functional Programming**
Well-known technique of efficient programming of (sometimes inefficient) programs

**Combination:**
Efficient compile-time resource analysis for functional code as shown in our earlier work (POPL’03)

**TODAY:** Application to object-oriented programming style (ESOP’06)
The Result:

Efficient compile-time resource analysis for (simplified) JAVA, successfully treating:
  • inheritance
  • downcast (and upcast)
  • imperative field update
  • aliasing (and circular data structures)

We have neglected:
  • multiple ancestors
  • exception handling
  • static classes
  • full inference of enriched types
Amortised Analysis

Example: Simulating queue (FIFO) by two stacks (LIFO)

Always push onto A and pop from B

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>⊠</td>
<td>⊠</td>
<td>⊠</td>
</tr>
<tr>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>

| op | push(●) | pop() = ⊗ | pop() = □ |
Amortised Analysis

Example: Simulating queue (FIFO) by two stacks (LIFO)

Always push onto A and pop from B

<table>
<thead>
<tr>
<th>op</th>
<th>push(◯)</th>
<th>pop() = ⊗</th>
<th>pop() = □</th>
</tr>
</thead>
<tbody>
<tr>
<td>cost</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>
Amortised Analysis

Example: Simulating queue (FIFO) by two stacks (LIFO)

Always push onto A and pop from B

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

op  | push(◯) | pop() = ⊗ | pop() = □ |
---|---|---|---|
cost | 1 | 1 | 5 |
**Amortised Analysis**

Example: Simulating queue (FIFO) by two stacks (LIFO)

Always push onto A and pop from B

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>![circle]</td>
<td>![square]</td>
<td>![checkmark]</td>
<td>![circle]</td>
<td>![square]</td>
</tr>
<tr>
<td>![circle]</td>
<td>![square]</td>
<td>![checkmark]</td>
<td>![circle]</td>
<td>![square]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Φ</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>cost</th>
<th>push(◯)</th>
<th>pop() = ⊗</th>
<th>pop() = □</th>
</tr>
</thead>
<tbody>
<tr>
<td>ΔΦ</td>
<td></td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Σ</td>
<td></td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Amortised costs are constant as opposed to actual cost!
Automated Analysis of Functional Code: Idea

- Assign potential to data based on type
  Type constructors receive weights \((\text{list}(\text{int}, 0), \text{list}(\text{int}, 1), \ldots)\)
  Functions receive weights \((\text{list}(\text{int}, 4) \xrightarrow{8/2} \text{list}(\text{int}, 0), \ldots)\)
- Abstract from actual values \((\text{list}(\text{int}, x), \text{list}(\text{int}, y), \ldots)\)
- Gather constraints from type derivation with amortised costs
- Feed constraints to LP solver

Successful heap-space analysis of first-order functional programs
applied in EU FET-IST project Mobile Resource Guarantees

Extended to higher-order functional programs meanwhile
currently applied in EU FET-IST project EmBounded
Automated Analysis of Functional Code: Idea

- Assign potential to data based on type
  - Type constructors receive weights \((\text{list}(\text{int}, 0), \text{list}(\text{int}, 1), \ldots)\)
  - Functions receive weights \((\text{list}(\text{int}, 4) \xrightarrow{8/2} \text{list}(\text{int}, 0), \ldots)\)
- Abstract from actual values \((\text{list}(\text{int}, x), \text{list}(\text{int}, y), \ldots)\)
- Gather constraints from type derivation with amortised costs
- Feed constraints to LP solver

Successful heap-space analysis of first-order functional programs applied in EU FET-IST project Mobile Resource Guarantees

Extended to higher-order functional programs meanwhile currently applied in EU FET-IST project EmBounded
Automated Analysis of Functional Code: Idea

- Assign potential to data based on type
  Type constructors receive weights \((\text{list}(\text{int}, 0), \text{list}(\text{int}, 1), \ldots)\)
  Functions receive weights \((\text{list}(\text{int}, 4) \xrightarrow{8/2} \text{list}(\text{int}, 0), \ldots)\)

- Abstract from actual values \((\text{list}(\text{int}, x), \text{list}(\text{int}, y), \ldots)\)

- Gather constraints from type derivation with amortised costs

- Feed constraints to LP solver

Successful heap-space analysis of first-order functional programs applied in EU FET-IST project Mobile Resource Guarantees

Extended to higher-order functional programs meanwhile currently applied in EU FET-IST project EmBounded
Automated Analysis of Functional Code: Idea

- Assign potential to data based on type
  Type constructors receive weights (list(int, 0), list(int, 1), ...)
  Functions receive weights (list(int, 4) $\xrightarrow{8/2}$ list(int, 0), ...)

- Abstract from actual values
  (list(int, $x$), list(int, $y$), ...)

- Gather constraints from type derivation with amortised costs

- Feed constraints to LP solver

Successful heap-space analysis of first-order functional programs applied in EU FET-IST project Mobile Resource Guarantees

Extended to higher-order functional programs meanwhile currently applied in EU FET-IST project EmBounded
Automated Analysis of Functional Code: Result

\[ f : \text{list(list(int, 1), 2.3)} \xrightarrow{4/6} \text{list(int, 5)} \]

Evaluating \( f([l_1, \ldots, l_m]) \)

- requires at most \( 4 + 2.3m + 1\Sigma|l_i| \) extra heap units and
- leaves at least \( 6 + 5|f(l)| \) unused memory units

Potential of consumed input furnishes upper bound on overall heap-consumption at runtime — without any runtime mechanics!

Annotations are weight factors — *no* reference to length/size

as *opposed* to sized types [Hughes & Pareto ’99,’02]
Amortised Analysis of Heap-Usage for OOP

- Types assign each heap configuration statically a potential

- Any object creation must be paid for, using the potential of input consumed

- Potential of consumed input furnishes upper bound on overall heap space consumption of program – no work at runtime!
Object-Oriented Language: RAJA

\[
c ::= \text{class } C \ [\text{extends } D] \ \{A_1; \ldots; A_k; M_1 \cdots M_j\}
\]
\[
A ::= C \ a
\]
\[
M ::= C_0 \ m(C_1 \ x_1, \ldots, C_j \ x_j)\{\text{return } e;\}
\]
\[
e ::= x \quad \text{(Variable)}
\]
\[
\mid \text{null} \quad \text{(Constant)}
\]
\[
\mid \text{new } C \quad \text{(Construction)}
\]
\[
\mid \text{free}(x) \quad \text{(Destruction)}
\]
\[
\mid (C)x \quad \text{(Cast)}
\]
\[
\mid x.a_i \quad \text{(Access)}
\]
\[
\mid x.a_i= x \quad \text{(Update)}
\]
\[
\mid x.m(x_1,\ldots,x_j) \quad \text{(Invocation)}
\]
\[
\mid \text{if } x \text{ instanceof } C \text{ then } e_1 \text{ else } e_2 \quad \text{(Conditional)}
\]
\[
\mid \text{let } x = e_1 \text{ in } e_2 \quad \text{(Let)}
\]

\approx \text{Featherweight Java (Igarashi, Pierce, Wadler; OOPSLA’99)}\]
\[
\text{plus} \text{ imperative field update}
\]
Memory Model

Similar to **Storeless Semantics** (Jonkers; Rinetzky, Wilhelm et al)

- captures quantities and aliasing
- no random reanimation of stale pointers
  
  ("Alias Types" Morrisett & Walker)
  ("Bunched Implication Logic" Ishtiaq & O’Hearn)

Free-list based model

- memory units taken from free-list at object creation
- memory units returned to free-list at object destruction
- deallocation in C/C++ style with primitive dispose
- dereferencing dangling pointers leads to abortion

Our goal: infer an upper bound on the size of the free-list required to successfully evaluate as function of the input
Amortised Typing

We use a typing judgement of the form

$$\Gamma \vdash_{m \to m'} e : C$$

meaning that if $E, h \vdash e \leadsto v, h'$ then a freelist whose size exceeds

$$m + \sum_{x : \text{dom}(\Gamma)} POTENTIAL_h(E(x) : \Gamma(x))$$

will suffice for successful evaluation and the freelist size upon completion will exceed $m' + POTENTIAL_{h'}(v : A)$. 
Amortised Typing

We use a typing judgement of the form $\Gamma \vdash \frac{m}{m'} e : C$

**Intuition:**

- $m$ is like cash in your pocket, ready to be spent, whereas
- $\Gamma$ is like money on the bank that you have to withdraw first

**Recall:**

$$m + \sum_{x : \text{dom}(\Gamma)} \text{POTENTIAL}_h(E(x) : \Gamma(x))$$
Typing Rule for Object Creation

\[ \varnothing \vdash^{p + \text{Size}(C)}_{0} \text{new } C : C \]

In a method call we get access to the annotation of the callee:

\[
\text{this:}C, \ x_1:A_1, \ldots, \ x_n:A_n \vdash^{m+p}_{m'} \ e_f:B
\]

then method \( f \) in class \( C \) with body \( e_f \) may be typed as

\[
B, m' \ f(A_1 \ x_1, \ldots, \ A_n \ x_n, m)
\]

\( p \) must depend on \( C \), its superclasses and its fields somehow
Reclaiming Potential

Q: Why can potential be spent without destroying objects?
A: Reclaiming potential *only* at object destruction would not be sufficient; non-destructively processing a data structure might need potential (e.g. clone) See example.

Q: Can you ‘gain’ potential without actually calling a method?
A: No. “this” is the only certain non-null pointer. A language with a separate category of non-null pointers would allow it.

Q: Will multiple calls to a method not mess up the potential?
A: Our sharing rules* will ensure that the second time around the callee has a different type which carries less if any potential.

*aka contraction
Sketch of RAJA System

RAJA program $P$ consists of a set of views. For each class $C$ and view $r$ we have an annotated version $C^r$.

\[
\diamond (C^r) : \text{Class} \times \text{View} \rightarrow \mathbb{Q}^+ \\
A^{\text{get}}(C^r, a) : \text{Class} \times \text{View} \times \text{Field} \rightarrow \text{View} \quad \text{(get-view)} \\
A^{\text{set}}(C^r, a) : \text{Class} \times \text{View} \times \text{Field} \rightarrow \text{View} \quad \text{(set-view)} \\
M(C^r, m) : \text{Class} \times \text{View} \times \text{Method} \rightarrow \\
\mathcal{P}(\text{Views of Arguments} \rightarrow \text{Effect} \times \text{View of Result})
\]

Subtyping of annotated classes is covariant w.r.t. $\diamond(\cdot)$, $A^{\text{get}}(\cdot, \cdot)$ and result types of methods and it is contravariant w.r.t. $A^{\text{set}}(\cdot, \cdot)$ and argument types of methods

\[
r_1, \ldots, r_j \xrightarrow{p/q} r_0
\]
Example: OO-Lists

abstract class List { abstract List clone(); }

class Nil extends List {
    List clone() {
        return this;
    }
}

class Cons extends List {
    Int elem;
    List next;

    List clone() {
        Cons res = new Cons();
        res.elem = this.elem;
        res.next = this.next.clone();
        return res;
    }
}

\[
\begin{array}{c|ccc}
\diamond() & \text{rich} & \text{poor} & \text{poorest} \\
\hline
\text{List} & 0 & 0 & 0 \\
\text{Nil} & 0 & 0 & 0 \\
\text{Cons} & 1 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|ccc}
\text{A}_{\text{get}}(\text{Cons}^x, \text{next}) & \text{rich} & \text{poor} & \text{poorest} \\
\text{A}_{\text{set}}(\text{Cons}^x, \text{next}) & \text{rich} & \text{poor} & \text{rich} \\
\end{array}
\]

\[
M(\{\text{List}^{\text{rich}}, \text{Cons}^{\text{rich}}, \text{Nil}^{\text{rich}}\}, \text{copy}) = () \xrightarrow{0/0} \text{poor}
\]
Potential
\[ \Phi_\sigma(v : r) = \sum_{\vec{p}} \phi_\sigma((v:r).\vec{p}) \]

Potential: infinite sum over all access paths from an object \( v \), zero almost everywhere (allowing cyclic data structures)

\[ \phi_\sigma((v:r).\vec{p}) = \begin{cases} 
0 & \text{if } [v.\vec{p}]_\sigma = \text{NULL or undefined} \\
\diamondsuit(D^s) & \text{otherwise}
\end{cases} \]

where \( D \) is the dynamic class type of \( v.\vec{p} \) and \( s \) is the view obtained by chaining \( r \) through the various dynamic types encountered starting from \( v \) along \( \vec{p} \) using \( A^{\text{get}} \)

- value \( v \): location or NULL
- view \( r \): obtained from static typing of \( v \)
- access path \( \vec{p} \): finite word over field names
- heap \( \sigma \): maps locations to objects

\[ \diamondsuit(\cdot) | \begin{array}{ccc}
\text{rich} & \text{poor} & \text{poorest} \\
\text{List} & 0 & 0 & 0 \\
\text{Nil} & 0 & 0 & 0 \\
\text{Cons} & 1 & 0 & 0 \\
\end{array} \]

\[ A^{\text{get}}(\text{Cons}^x, \text{next}) | \begin{array}{ccc}
\text{rich} & \text{poor} & \text{poorest} \\
\text{rich} & \text{poor} & \text{poorest} \\
\text{rich} & \text{poor} & \text{rich} \\
\end{array} \]

\[ A^{\text{set}}(\text{Cons}^x, \text{next}) | \begin{array}{ccc}
\text{rich} & \text{poor} & \text{poorest} \\
\text{rich} & \text{poor} & \text{rich} \\
\text{rich} & \text{poor} & \text{rich} \\
\end{array} \]

\[ M\left(\{\text{List}^{\text{rich}}, \text{Cons}^{\text{rich}}, \text{Nil}^{\text{rich}}\}, \text{copy}\right) = () \xrightarrow{0/0} \text{poor} \]
Example: OO-Lists

$v$ points to chain of 3 $\text{Cons}$ objects followed by a $\text{Nil}$ object in $\sigma$

\[
\begin{align*}
\phi_\sigma((v:\text{rich}).\epsilon) &= \Diamond (\text{Cons}^{\text{rich}}) = 1 \\
\phi_\sigma((v:\text{rich}).\text{next}) &= 1 \\
\phi_\sigma((v:\text{rich}).\text{next}.\text{next}) &= 1 \\
\phi_\sigma((v:\text{rich}).\text{next}.\text{next}.\text{next}) &= \Diamond (\text{Nil}^{\text{rich}}) = 0 \\
\phi_\sigma((v:\text{rich}).\text{next}.\text{next}.\text{next}.\text{next}.\ast) &= 0
\end{align*}
\]

Therefore:

\[
\Phi_\sigma(v : \text{rich}) = 3 \quad \text{but} \quad \Phi_\sigma(v : \text{poor}) = 0
\]

<table>
<thead>
<tr>
<th>$\Diamond(\cdot)$</th>
<th>rich</th>
<th>poor</th>
<th>poorest</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Nil</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cons</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

$\text{A}_{\text{get}}(\text{Cons}^x, \text{next})$

$\text{A}_{\text{set}}(\text{Cons}^x, \text{next})$

$M(\{\text{List}^{\text{rich}}, \text{Cons}^{\text{rich}}, \text{Nil}^{\text{rich}}\}, \text{copy}) = () \xrightarrow{0/0} \text{poor}$
**RAJA Typing Rules**

- Upon object creation (**new**) one must pay the actual cost (size of the object) *and also* the amortised cost (e.g. $+1$ in the case of $\text{Cons}^{\text{rich}}$)

- In the body of a method one gets access to the annotation of the callee, however it must be shared with possible uses of **this** in the method body, see below.

- In a deallocation (**free**) one gets access to both the annotation and the actual size of the object.

- To prevent multiple access to annotations via multiple method calls, we use a linear typing discipline with an explicit contraction rule (**sharing**):
### Aliasing

\[
\forall (s \mid q_1, q_2) \quad \exists \, \gamma \vdash D^{q_1}, z : D^{q_2} \quad \frac{n}{n'} e : C^r \\
\frac{}{\exists \, \Gamma, x : D^s \quad \frac{n}{n'} e[x/y, x/z] : C^r}
\]

\(\forall (\cdot | \cdot)\): coinductively defined relation between views and multisets of views.

We do have:

\[\gamma(\text{poor} \mid \{\text{poor, poor, poor, ...}\})\]
\[\gamma(\text{rich} \mid \{\text{rich, poorest, poorest, ...}\})\]

Of course, we do not have:

\[\gamma(\text{poor} \mid \{\text{rich, poor}\})\]
\[\gamma(\text{rich} \mid \{\text{rich, rich}\})\]
\[\gamma(\text{rich} \mid \{\text{rich, poor}\})\]

<table>
<thead>
<tr>
<th>(\diamond (\cdot))</th>
<th>rich</th>
<th>poor</th>
<th>poorest</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Nil</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cons</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[M(\{\text{List}^{\text{rich}}, \text{Cons}^{\text{rich}}, \text{Nil}^{\text{rich}}\}, \text{copy}) = () \rightarrow \text{poor}\]
**Update Rule**

\[
A^{\text{set}}(C^r, a) = s \quad C.a = D \\
\frac{x:C^r, y:D^s}{0} \quad x.a \leftarrow y : C^r
\]

Field update requires a view which is rich enough to feed all different paths that might lead into this field.

Thus, if \( x \) has type \( \text{List}^{\text{poorest}} \) then for

\[
x.\text{next} \leftarrow y;
\]

one must have \( y: \text{List}^{\text{rich}} \).

After all, the above code could have been preceded by

\[
x = z;
\]

with \( z: \text{List}^{\text{rich}} \), then after the assignment we would still expect \( z \) to be “rich” and fortunately it is!

However, even \( x.\text{next} \leftarrow x.\text{next}; \) is now forbidden.
**Update Rule**

\[
A^\text{set}(C^r, a) = s \quad C.a = D
\]

\[
x : C^r, y : D^s \mid \overline{0} \quad x.a \leftarrow y : C^r
\]

Our rule differs from standard Java field update:

\[
C.a = D
\]

\[
x : C, y : D \mid x.a \leftarrow y : D
\]

Java-style update is definable:

\[
\text{let } u = (x.a \leftarrow y) \text{ in } y
\]

but relies on sharing as it should be!
Soundness Theorem

If \( \Gamma \vdash^{n}_{n'} e : C^r \), \( \eta, \sigma \vdash e \sim v, \tau \), and \( \sigma \models \eta : (\Gamma, \Delta) \) then

\[
\eta, \sigma \vdash^{\frac{n}{n'} + \Phi_{\sigma}(\eta : \Gamma) + \Phi_{\sigma}(\eta : \Delta)}_{\frac{n'}{n'} + \Phi_{\tau}(v : r) + \Phi_{\tau}(\eta : \Delta)} e \sim v, \tau \tag{1}
\]

\[
\tau \models \eta[x_{res} \mapsto v] : (\Delta, x_{res} : C^r) \tag{2}
\]

\( \Delta \) is an arbitrary context representing other parts of the program that may share with the currently focused on heap portion.

The statement of the soundness theorem is similar to [HJ 2003].
Proof sketch: Update
Suppose we deal with the update expression \( x.\text{next} \leftarrow y \)

In the worst case we had before \([x.\text{next}]_\sigma = \text{NULL},\)
i.e. no potential contributed by paths containing \( x.\text{next} \)
Proof sketch: Update
Suppose we deal with the update expression \( x.\text{next} \leftarrow y \)

In the worst case we had before \( [x.\text{next}]_\sigma = \text{NULL} \), i.e. no potential contributed by paths containing \( x.\text{next} \)

Now each access path arriving at \( x \) can continue through \( \text{next} \) and any path leading away from \( y \) — possibly contributing potential.
Proof sketch: Update
Suppose we deal with the update expression \( x.\text{next} \leftarrow y \)

In the worst case we had before \( [x.\text{next}]_\sigma = \text{NULL} \), i.e. no potential contributed by paths containing \( x.\text{next} \)

Now each access path arriving at \( x \) can continue through \( \text{next} \) and any path leading away from \( y \) – possibly contributing potential.

We proof that there is no unsound increase in potential!
More examples

- **Doubly-linked lists:** even in rich version the back-pointers are poorest so that only access paths of the form `next*` contribute.

- **Iterators on doubly linked lists:** as soon as you move the iterator backwards it changes view so no more potential can be extracted.

Planned examples: visitor, subject-observer, union-find.
Conclusion

• Our type-based analysis encompasses:
  ★Objects ★Inheritance ★Downcast
  ★Imperative Update ★Aliasing ★Circular Data

• Type inference nontrivial task. Tree automata?

• More examples and implementation are being worked on.

• Applicable to other quantitative properties:
  number of calls to methods other than “new”, e.g. “fopen”,
  or stack-size, execution time, ...