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)

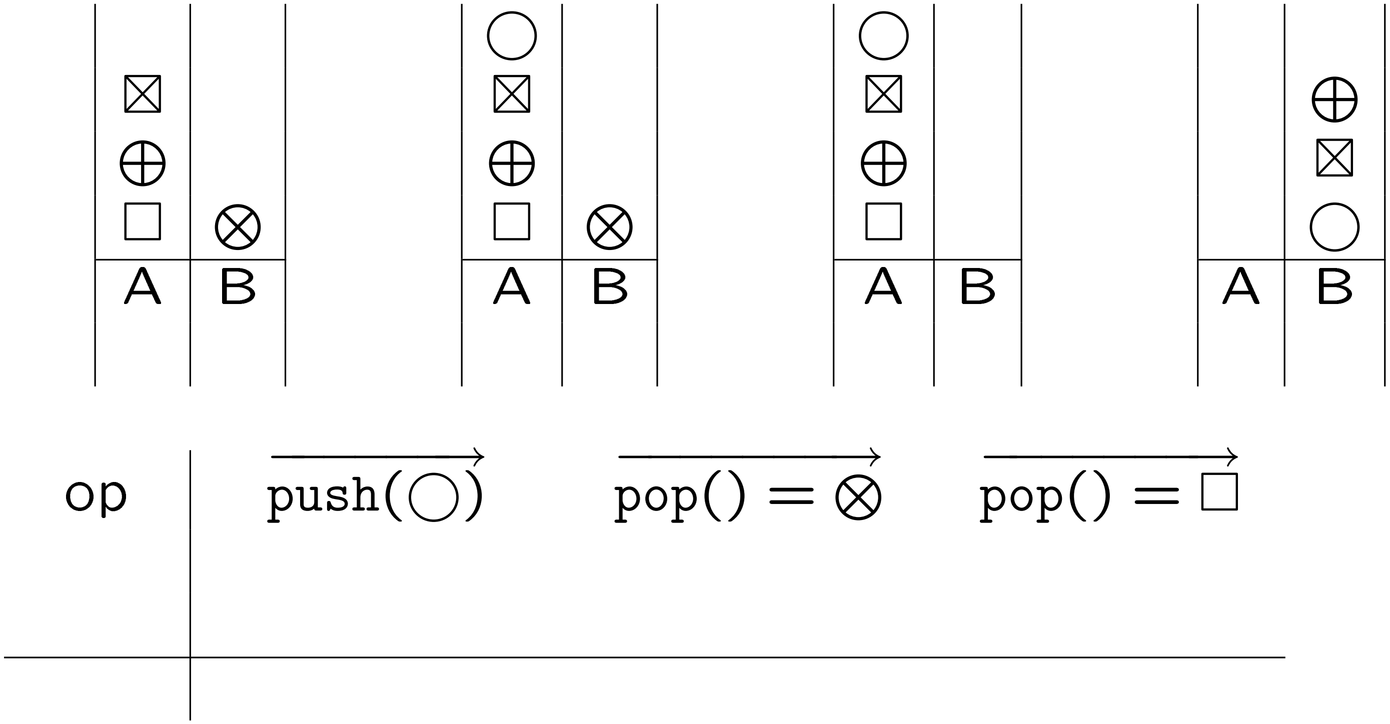
We have neglected:

- 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

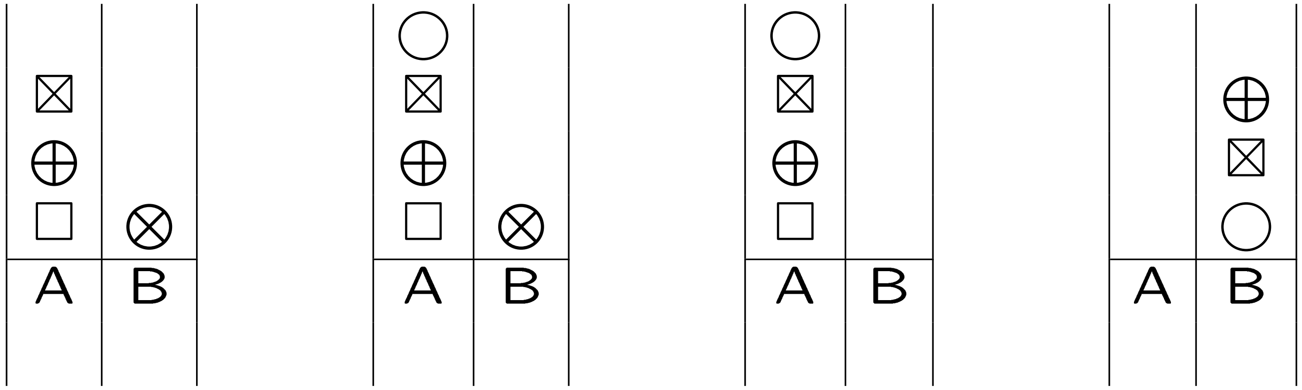


$\overrightarrow{\texttt{push}(\bigcirc)}$ $\overrightarrow{\texttt{pop}() = \otimes}$ $\overrightarrow{\texttt{pop}() = \square}$

# Amortised Analysis

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

Always push onto A and pop from B

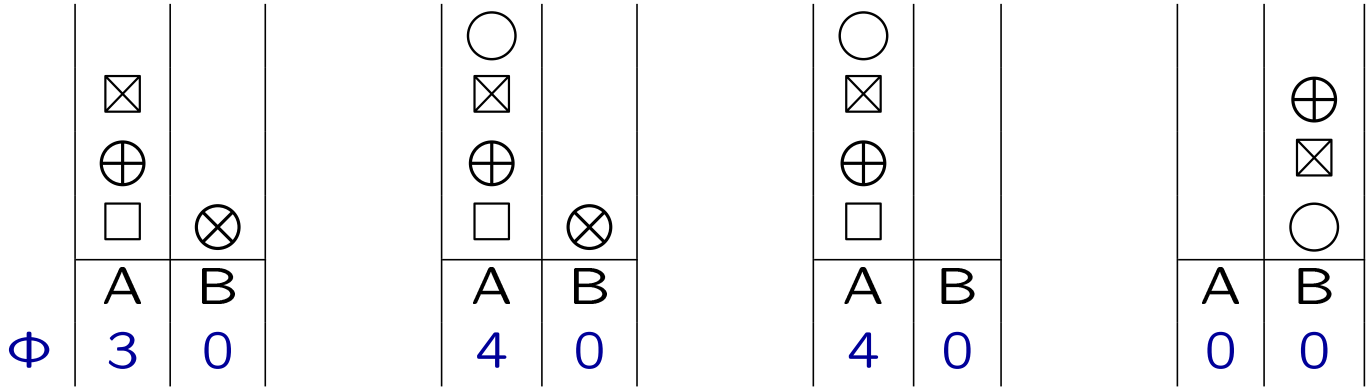


|  |  |  |  |
|---|---|---|---|
| op | $\overrightarrow{\mathtt{push}(\bigcirc)}$ | $\overrightarrow{\mathtt{pop}() = \otimes}$ | $\overrightarrow{\mathtt{pop}() = \square}$ |
| cost | 1 | 1 | 5 |

# Amortised Analysis

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

Always push onto A and pop from B

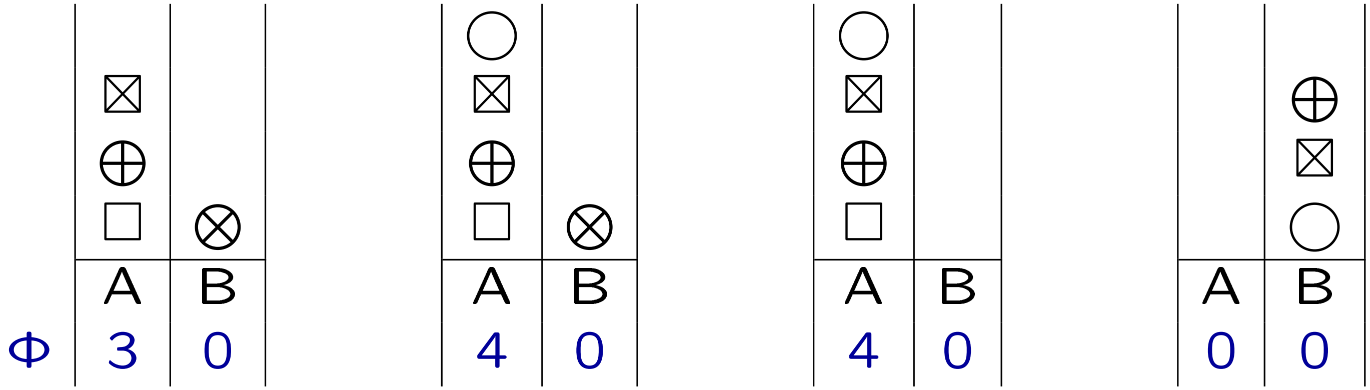


| op | $\overrightarrow{\texttt{push}(\bigcirc)}$ | $\overrightarrow{\texttt{pop}() = \otimes}$ | $\overrightarrow{\texttt{pop}() = \Box}$ |
|---|---|---|---|
| cost | 1 | 1 | 5 |

## Amortised Analysis

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

Always push onto A and pop from B



| op | $\overrightarrow{\mathtt{push}(\bigcirc)}$ | $\overrightarrow{\mathtt{pop}() = \otimes}$ | $\overrightarrow{\mathtt{pop}() = \square}$ |
|---|---|---|---|
| cost | 1 | 1 | 5 |
| $\Delta\Phi$ | 1 | 0 | $-4$ |
| $\Sigma$ | 2 | 1 | 1 |

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 $(\mathsf{list}(\mathsf{int}, 0), \mathsf{list}(\mathsf{int}, 1), \ldots)$

  Functions receive weights $(\mathsf{list}(\mathsf{int}, 4) \xrightarrow{8/2} \mathsf{list}(\mathsf{int}, 0), \ldots)$
- Abstract from actual values $(\mathsf{list}(\mathsf{int}, x), \mathsf{list}(\mathsf{int}, y), \ldots)$
- 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 $(\mathsf{list}(\mathsf{int}, 0), \mathsf{list}(\mathsf{int}, 1), \ldots)$
  Functions receive weights $(\mathsf{list}(\mathsf{int}, 4) \xrightarrow{8/2} \mathsf{list}(\mathsf{int}, 0), \ldots)$
- Abstract from actual values $(\mathsf{list}(\mathsf{int}, x), \mathsf{list}(\mathsf{int}, y), \ldots)$
- 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 $(\mathsf{list}(\mathsf{int}, 0), \mathsf{list}(\mathsf{int}, 1), \dots)$
  Functions receive weights $(\mathsf{list}(\mathsf{int}, 4) \xrightarrow{8/2} \mathsf{list}(\mathsf{int}, 0), \dots)$
- Abstract from actual values $(\mathsf{list}(\mathsf{int}, x), \mathsf{list}(\mathsf{int}, y), \dots)$
- 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 $(\mathsf{list}(\mathsf{int},0),\mathsf{list}(\mathsf{int},1),\dots)$
  Functions receive weights $(\mathsf{list}(\mathsf{int},4) \xrightarrow{8/2} \mathsf{list}(\mathsf{int},0),\dots)$
- Abstract from actual values $(\mathsf{list}(\mathsf{int},x),\mathsf{list}(\mathsf{int},y),\dots)$
- 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

$$\mathtt{f} : \mathsf{list}(\mathsf{list}(\mathsf{int}, 1), 2.3) \xrightarrow{4/6} \mathsf{list}(\mathsf{int}, 5)$$

Evaluating $\mathtt{f}\Big([l_1, \ldots, l_m]\Big)$

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

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

≈ Featherweight Java (Igarashi, Pierce, Wadler; OOPSLA'99)
*plus* 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 primitve 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

$$\Gamma \mathrel{\Big|\!\frac{m}{m'}} e : C$$

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

$$m + \sum_{x\,:\,\mathsf{dom}(\Gamma)} \mathit{POTENTIAL}_h(E(x) : \Gamma(x))$$

will suffice for successful evaluation and the freelist size upon completion will exceed $m' + \mathit{POTENTIAL}_{h'}(v : A)$.

# Amortised Typing

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

*Intuition:*

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

*Recall:*

$$m + \sum_{x \colon \mathsf{dom}(\Gamma)} \mathit{POTENTIAL}_h(E(x) : \Gamma(x))$$

# Typing Rule for Object Creation

$$\frac{}{\varnothing \;\vdash^{p + \mathsf{Size}(C)}_{0}\; \mathtt{new}\ C : C}$$

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

$$\mathtt{this}{:}C,\ x_1{:}A_1,\ \ldots,\ x_n{:}A_n \;\vdash^{m+p}_{m'}\; e_f{:}B$$

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 $C^r$.

$$
\begin{aligned}
\Diamond(C^r) &: \text{Class} \times \text{View} \to \mathbb{Q}^+ \\
\mathsf{A}^{\text{get}}(C^r, a) &: \text{Class} \times \text{View} \times \text{Field} \to \text{View} \quad \text{(get-view)} \\
\mathsf{A}^{\text{set}}(C^r, a) &: \text{Class} \times \text{View} \times \text{Field} \to \text{View} \quad \text{(set-view)} \\
\mathsf{M}(C^r, m) &: \text{Class} \times \text{View} \times \text{Method} \to \\
&\qquad \mathcal{P}(\text{Views of Arguments} \to \text{Effect} \times \text{View of Result})
\end{aligned}
$$

$$r_1, \ldots, r_j \xrightarrow{p/q} r_0$$

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

# 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; }
}
```

| $\Diamond(\cdot)$ | rich | poor | poorest |
|---|---|---|---|
| `List` | 0 | 0 | 0 |
| `Nil` | 0 | 0 | 0 |
| `Cons` | 1 | 0 | 0 |

| | rich | poor | poorest |
|---|---|---|---|
| $\mathsf{A}^{\mathsf{get}}(\mathtt{Cons}^{x}, \mathtt{next})$ | rich | poor | poorest |
| $\mathsf{A}^{\mathsf{set}}(\mathtt{Cons}^{x}, \mathtt{next})$ | rich | poor | rich |

$$\mathsf{M}\big(\{\mathtt{List}^{\mathsf{rich}}, \mathtt{Cons}^{\mathsf{rich}}, \mathtt{Nil}^{\mathsf{rich}}\}, \mathtt{copy}\big) = () \xrightarrow{0/0} \mathsf{poor}$$

## Potential

$$\Phi_\sigma(v : r) = \sum_{\vec{p}} \phi_\sigma\big((v{:}r).\vec{p}\big)$$

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

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

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 $\mathsf{A}^{\mathsf{get}}$

| | |
|---|---|
| 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 |

| $\Diamond(\cdot)$ | rich | poor | poorest |
|---|---|---|---|
| `List` | 0 | 0 | 0 |
| `Nil` | 0 | 0 | 0 |
| `Cons` | 1 | 0 | 0 |

| | rich | poor | poorest |
|---|---|---|---|
| $\mathsf{A}^{\mathsf{get}}(\texttt{Cons}^x, \texttt{next})$ | rich | poor | poorest |
| $\mathsf{A}^{\mathsf{set}}(\texttt{Cons}^x, \texttt{next})$ | rich | poor | rich |

$$\mathsf{M}\big(\{\texttt{List}^{\mathsf{rich}}, \texttt{Cons}^{\mathsf{rich}}, \texttt{Nil}^{\mathsf{rich}}\}, \texttt{copy}\big) = () \xrightarrow{0/0} \mathsf{poor}$$

# Example: OO-Lists

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

$$
\begin{aligned}
&\phi_\sigma\Big((v{:}\mathsf{rich}).\epsilon\Big) = \Diamond\Big(\texttt{Cons}^{\mathsf{rich}}\Big) = 1\\
&\phi_\sigma\Big((v{:}\mathsf{rich}).\texttt{next}\Big) = 1\\
&\phi_\sigma\Big((v{:}\mathsf{rich}).\texttt{next.next}\Big) = 1\\
&\phi_\sigma\Big((v{:}\mathsf{rich}).\texttt{next.next.next}\Big) = \Diamond\Big(\texttt{Nil}^{\mathsf{rich}}\Big) = 0\\
&\phi_\sigma\Big((v{:}\mathsf{rich}).\texttt{next.next.next.next}^*\Big) = 0
\end{aligned}
$$

Therefore:

$$\Phi_\sigma(v : \mathsf{rich}) = 3 \qquad \text{but} \qquad \Phi_\sigma(v : \mathsf{poor}) = 0$$

| $\Diamond(\cdot)$ | rich | poor | poorest |
|---|---|---|---|
| `List` | 0 | 0 | 0 |
| `Nil` | 0 | 0 | 0 |
| `Cons` | 1 | 0 | 0 |

| | rich | poor | poorest |
|---|---|---|---|
| $\mathsf{A}^{\mathsf{get}}(\texttt{Cons}^x, \texttt{next})$ | rich | poor | poorest |
| $\mathsf{A}^{\mathsf{set}}(\texttt{Cons}^x, \texttt{next})$ | rich | poor | rich |

$$\mathsf{M}\Big(\{\texttt{List}^{\mathsf{rich}}, \texttt{Cons}^{\mathsf{rich}}, \texttt{Nil}^{\mathsf{rich}}\}, \texttt{copy}\Big) = () \xrightarrow{0/0} \mathsf{poor}$$

# 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 $\texttt{Cons}^{\mathsf{rich}}$)
- 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

$$\frac{\curlyvee(s \,|\, q_1, q_2) \qquad \Gamma, y{:}D^{q_1}, z{:}D^{q_2} \vdash^{n}_{n'} e : C^{r}}{\Gamma, x{:}D^{s} \vdash^{n}_{n'} e[x/y, x/z] : C^{r}}$$

$\curlyvee(\cdot \,|\, \cdot)$: coinductively defined relation between views and multisets of views.

We do have:

$\curlyvee(\text{poor} \,|\, \{\text{poor}, \text{poor}, \text{poor}, \ldots\})$

$\curlyvee(\text{rich} \,|\, \{\text{rich}, \text{poorest}, \text{poorest}, \ldots\})$

Of course, we do *not* have:

$\curlyvee(\text{poor} \,|\, \{\text{rich}, \text{poor}\})$

$\curlyvee(\text{rich} \,|\, \{\text{rich}, \text{rich}\})$

$\curlyvee(\text{rich} \,|\, \{\text{rich}, \text{poor}\})$

| $\Diamond(\cdot)$ | rich | poor | poorest |
|---|---|---|---|
| `List` | 0 | 0 | 0 |
| `Nil` | 0 | 0 | 0 |
| `Cons` | 1 | 0 | 0 |

| | rich | poor | poorest |
|---|---|---|---|
| $\mathsf{A}^{\mathsf{get}}(\mathtt{Cons}^{x}, \mathtt{next})$ | rich | poor | poorest |
| $\mathsf{A}^{\mathsf{set}}(\mathtt{Cons}^{x}, \mathtt{next})$ | rich | poor | rich |

$$\mathsf{M}\big(\{\mathtt{List}^{\mathsf{rich}}, \mathtt{Cons}^{\mathsf{rich}}, \mathtt{Nil}^{\mathsf{rich}}\}, \mathtt{copy}\big) = () \xrightarrow{0/0} \text{poor}$$

# Update Rule

$$\frac{\mathsf{A}^{\mathsf{set}}(C^{r}, a) = s \qquad\qquad C.a = D}{x{:}C^{r}, y{:}D^{s} \mathrel{\vdash^{0}_{0}} x.a\texttt{<-}y : C^{r}}$$

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 `List`$^{\text{poorest}}$ then for

```
x.next<-y;
```

one must have `y:List`$^{\text{rich}}$.

After all, the above code could have been preceded by

```
x = z;
```

with `z:List`$^{\text{rich}}$, then after the assignment we would still expect `z` to be "`rich`" and fortunately it is!

However, even `x.next<-x.next;` is now forbidden.

# Update Rule

$$\frac{\mathsf{A}^{\mathsf{set}}(C^r, a) = s \qquad C.a = D}{x{:}C^r, y{:}D^s \vdash^{0}_{0} x.a\texttt{<-}y : C^r}$$

Our rule differs from standard Java field update:

$$\frac{C.a = D}{x{:}C, y{:}D \vdash x.a\texttt{<-}y : D}$$

Java-style update is definable:

$$\texttt{let } u \texttt{ = } (x.a\texttt{<-}y) \texttt{ in } y$$

but relies on sharing as it should be!

# Soundness Theorem

If $\quad \Gamma \mathrel{\vert\!\frac{n}{n'}} e : C^{r} \quad \eta, \sigma \vdash e \rightsquigarrow v, \tau \quad \sigma \vDash \eta : (\Gamma, \Delta)$ then

$$\eta, \sigma \;\mathrel{\vert\!\frac{n + \Phi_{\sigma}(\eta : \Gamma) + \Phi_{\sigma}(\eta : \Delta)}{n' + \Phi_{\tau}(v : r) + \Phi_{\tau}(\eta : \Delta)}}\; e \rightsquigarrow v, \tau \tag{1}$$

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

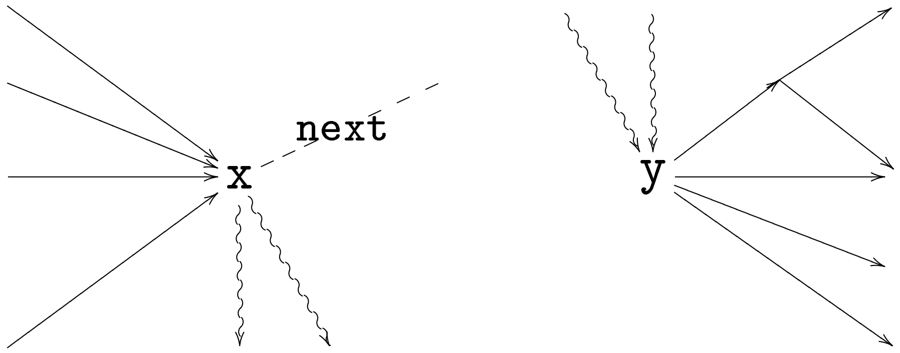
$\Delta$ is an arbitrary context representing other parts of the program that may share with the currently focused on heap portion.

The statement of the soundness theorem is similar to [HJ 2003].

# Proof sketch: Update

Suppose we deal with the update expression `x.next <- y`

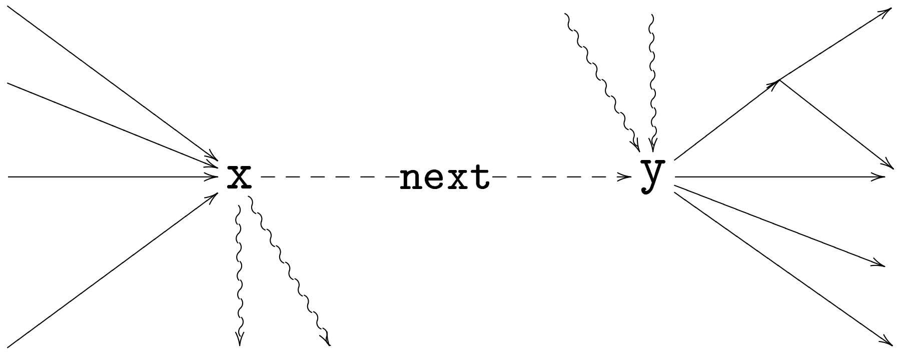
In the worst case we had before $[\![\texttt{x.next}]\!]_\sigma = \text{NULL}$,
i.e. no potential contributed by paths containing `x.next`



## Proof sketch: Update

Suppose we deal with the update expression `x.next <- y`

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

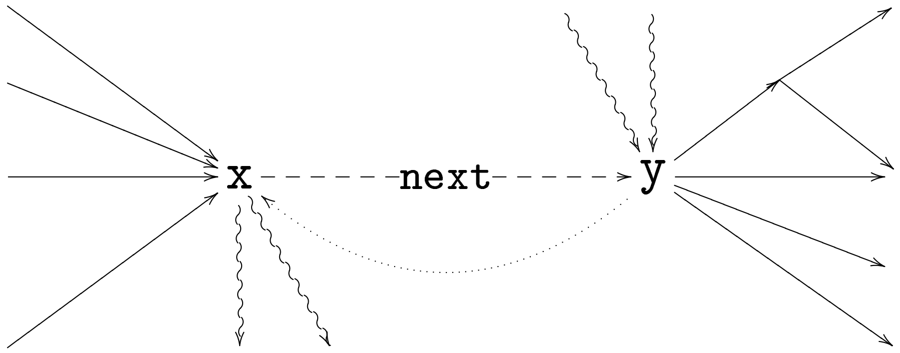


Now each access path arriving at `x` can continue through `next` and any path leading away from `y` – possibly contributing potential.

# Proof sketch: Update

Suppose we deal with the update expression `x.next <- y`

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



Now each access path arriving at `x` can continue through `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 $\mathtt{next}^*$ 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, . . .