

New Results on the Implementation of Resource Aware JAva (RAJA)

Dulma Rodriguez

Department of Computer Science
Ludwig-Maximilians-University Munich

4th Annual MOBIUS Meeting
June 18th, 2009



Introduction

- System RAJA (Hofmann and Jost, ESOP 2006)
 - Type system for Java-like programs.
 - Compile-time analysis of heap-space requirements.
 - Amortised complexity analysis.
- New results on the implementation
 - Elimination of some annotations in the programs.
 - Algorithmic typing rules (with correctness proof).
 - Algorithmic subtyping (with correctness proof).
 - Extension of the implementation with more practical features.

Introduction

- System RAJA (Hofmann and Jost, ESOP 2006)
 - Type system for Java-like programs.
 - Compile-time analysis of heap-space requirements.
 - Amortised complexity analysis.
- New results on the implementation
 - Elimination of some annotations in the programs.
 - Algorithmic typing rules (with correctness proof).
 - Algorithmic subtyping (with correctness proof).
 - Extension of the implementation with more practical features.

Outline

- 1 Introduction to RAJA
- 2 Algorithmic RAJA Typing
- 3 Algorithmic Subtyping
- 4 New Implementation Features
- 5 Conclusions

Amortised Analysis of Heap-Usage

- Free-list based model
 - Deallocation in C/C++ style with primitive dispose.
 - Object creation: takes memory units from free-list.
 - Object destruction: returns memory units to free-list.
- Goal:
 - Upper bound on the size of the free-list.
 - As function of the input.
- Amortised analysis
 - Types assign each heap configuration statically a potential.
 - Object creation: must pay using potential from input.
 - Potential of consumed input: upper bound on heap space consumption.

Amortised Analysis of Heap-Usage

- Free-list based model
 - Deallocation in C/C++ style with primitive dispose.
 - Object creation: takes memory units from free-list.
 - Object destruction: returns memory units to free-list.
- Goal:
 - Upper bound on the size of the free-list.
 - As function of the input.
- Amortised analysis
 - Types assign each heap configuration statically a potential.
 - Object creation: must pay using potential from input.
 - Potential of consumed input: upper bound on heap space consumption.

Amortised Analysis of Heap-Usage

- Free-list based model
 - Deallocation in C/C++ style with primitive dispose.
 - Object creation: takes memory units from free-list.
 - Object destruction: returns memory units to free-list.
- Goal:
 - Upper bound on the size of the free-list.
 - As function of the input.
- Amortised analysis
 - Types assign each heap configuration statically a potential.
 - Object creation: must pay using potential from input.
 - Potential of consumed input: upper bound on heap space consumption.

Object-Oriented Language: FJEU

$c ::=$	<code>class C [extends D] {A₁;...;A_k; M₁ ... M_j }</code>	
$A ::=$	<code>C a</code>	
$M ::=$	<code>C₀ m(C₁ x₁,..., C_j x_j) {return e; }</code>	
$e ::=$	<code>x</code>	(Variable)
	<code>null</code>	(Constant)
	<code>new C</code>	(Construction)
	<code>free(x)</code>	(Destruction)
	<code>(C)x</code>	(Cast)
	<code>x.a_i</code>	(Access)
	<code>x.a_i ← x</code>	(Update)
	<code>x.m(x₁,..., x_j)</code>	(Invocation)
	<code>if x instanceof C then e₁ else e₂</code>	(Conditional)
	<code>let D x = e₁ in e₂</code>	(Let)

- \approx Featherweight Java (Igarashi, Pierce, Wadler, OOPSLA'99) + imperative field update.

Copy example in RAJA

```
class List {
  List copy() {
    return null;
  }
}
class Nil extends List {
  List copy() {
    return this;
  }
}
```

```
class Cons extends List {
  int elem;
  List next;

  List copy() {
    let res = new Cons in
    let _ = res.elem ← this.elem in
    let rnext = this.next.copy() in
    let _ = res.next ← rnext in
    return res;
  }
}
```

Copy example in RAJA

```

class List {
    List copy() {
        return null;
    }
}
class Nil extends List {
    List copy() {
        return this;
    }
}

```

```

class Cons extends List {
    int elem;
    List next;

    List copy() {
        let res = new Cons in
        let _ = res.elem ← this.elem in
        let rnext = this.next.copy() in
        let _ = res.next ← rnext in
        return res;
    }
}

```

- Space consumption of $l.copy() : n = \text{length}(l)$.

Copy example in RAJA

```

views rich, poor
class List {
  rich, poor: pot = 0;
  rich: List<poor>,0 copy(0) {
    return null;
  }
}
class Nil extends List {
  rich, poor: pot = 0;
  rich: List<poor>,0 copy(0) {
    return this;
  } }

```

```

class Cons extends List {
  rich: pot = 1;
  poor: pot = 0;
  rich, poor: int elem;
  rich: List<rich,rich> next;
  poor: List<poor,poor> next;

  rich: List<poor>,0 copy(0) {
    let res = new Cons in
    let _ = res.elem ← this.elem in
    let rnext = this.next.copy() in
    let _ = res.next ← rnext in
    return res;
  } }

```

Copy example in RAJA

```

views rich, poor
class List {
  rich, poor: pot = 0;
  rich: List<poor>, 0 copy(0) {
    return null;
  }
}
class Nil extends List {
  rich, poor: pot = 0;
  rich: List<poor>, 0 copy(0) {
    return this;
  } }

```

```

class Cons extends List {
  rich: pot = 1;
  poor: pot = 0;
  rich, poor: int elem;
  rich: List<rich,rich> next;
  poor: List<poor,poor> next;

  rich: List<poor>, 0 copy(0) {
    let res = new Cons in
    let _ = res.elem ← this.elem in
    let rnext = this.next.copy() in
    let _ = res.next ← rnext in
    return res;
  } }

```

- $\diamond(List^{rich}) = \diamond(List^{poor}) = \diamond(Nil^{rich}) = \diamond(Nil^{poor}) = \diamond(Cons^{poor}) = 0$

Copy example in RAJA

```

views rich, poor
class List {
  rich, poor: pot = 0;
  rich: List<poor>,0 copy(0) {
    return null;
  }
}
class Nil extends List {
  rich, poor: pot = 0;
  rich: List<poor>,0 copy(0) {
    return this;
  } }

```

```

class Cons extends List {
  rich: pot = 1;
  poor: pot = 0;
  rich, poor: int elem;
  rich: List<rich,rich> next;
  poor: List<poor,poor> next;

  rich: List<poor>,0 copy(0) {
    let res = new Cons in
    let _ = res.elem ← this.elem in
    let rnext = this.next.copy() in
    let _ = res.next ← rnext in
    return res;
  } }

```

• $\diamond(List^{rich}) = \diamond(List^{poor}) = \diamond(Nil^{rich}) = \diamond(Nil^{poor}) = \diamond(Cons^{poor}) = 0 \quad \diamond(Cons^{rich}) = 1$

Copy example in RAJA

```

views rich, poor
class List {
  rich, poor: pot = 0;
  rich: List<poor>, 0 copy(0) {
    return null;
  }
}
class Nil extends List {
  rich, poor: pot = 0;
  rich: List<poor>, 0 copy(0) {
    return this;
  } }

```

```

class Cons extends List {
  rich: pot = 1;
  poor: pot = 0;
  rich, poor: int elem;
  rich: List<rich,rich> next;
  poor: List<poor,poor> next;

  rich: List<poor>, 0 copy(0) {
    let res = new Cons in
    let _ = res.elem ← this.elem in
    let rnext = this.next.copy() in
    let _ = res.next ← rnext in
    return res;
  } }

```

- $\diamond(List^{rich}) = \diamond(List^{poor}) = \diamond(Nil^{rich}) = \diamond(Nil^{poor}) = \diamond(Cons^{poor}) = 0$ $\diamond(Cons^{rich}) = 1$
- $A(Cons^{rich}, next) = (rich, rich)$

Copy example in RAJA

```

views rich, poor
class List {
  rich, poor: pot = 0;
  rich: List<poor>, 0 copy(0) {
    return null;
  }
}
class Nil extends List {
  rich, poor: pot = 0;
  rich: List<poor>, 0 copy(0) {
    return this;
  } }

```

```

class Cons extends List {
  rich: pot = 1;
  poor: pot = 0;
  rich, poor: int elem;
  rich: List<rich,rich> next;
  poor: List<poor,poor> next;

  rich: List<poor>, 0 copy(0) {
    let res = new Cons in
    let _ = res.elem ← this.elem in
    let rnext = this.next.copy() in
    let _ = res.next ← rnext in
    return res;
  } }

```

- $\diamond(List^{rich}) = \diamond(List^{poor}) = \diamond(Nil^{rich}) = \diamond(Nil^{poor}) = \diamond(Cons^{poor}) = 0$ $\diamond(Cons^{rich}) = 1$
- $A(Cons^{rich}, next) = (rich, rich)$ $A(Cons^{poor}, next) = (poor, poor)$

Copy example in RAJA

```

views rich, poor
class List {
  rich, poor: pot = 0;
  rich: List<poor>, 0 copy(0) {
    return null;
  }
}
class Nil extends List {
  rich, poor: pot = 0;
  rich: List<poor>, 0 copy(0) {
    return this;
  } }

```

```

class Cons extends List {
  rich: pot = 1;
  poor: pot = 0;
  rich, poor: int elem;
  rich: List<rich,rich> next;
  poor: List<poor,poor> next;

  rich: List<poor>, 0 copy(0) {
    let res = new Cons in
    let _ = res.elem ← this.elem in
    let rnext = this.next.copy() in
    let _ = res.next ← rnext in
    return res;
  } }

```

- $\diamond(List^{rich}) = \diamond(List^{poor}) = \diamond(Nil^{rich}) = \diamond(Nil^{poor}) = \diamond(Cons^{poor}) = 0$ $\diamond(Cons^{rich}) = 1$
- $A(Cons^{rich}, next) = (rich, rich)$ $A(Cons^{poor}, next) = (poor, poor)$
- $M(List^{rich}, copy) = M(Nil^{rich}, copy) = M(Cons^{rich}, copy) = () \xrightarrow{0/0} List^{poor}$

Copy example in RAJA

```

views rich, poor
class List {
  rich, poor: pot = 0;
  rich: List<poor>,0 copy(0) {
    return null;
  }
}
class Nil extends List {
  rich, poor: pot = 0;
  rich: List<poor>,0 copy(0) {
    return this;
  } }

```

```

class Cons extends List {
  rich: pot = 1;
  poor: pot = 0;
  rich, poor: int elem;
  rich: List<rich,rich> next;
  poor: List<poor,poor> next;

  rich: List<poor>,0 copy(0) {
    let res = new Cons in
    let _ = res.elem ← this.elem in
    let rnext = this.next.copy() in
    let _ = res.next ← rnext in
    return res;
  } }

```

- Space consumption of $l.copy()$: $n = \text{length}(l)$.

Copy example in RAJA

```

views rich, poor
class List {
  rich, poor: pot = 0;
  rich: List<poor>, 0 copy(0) {
    return null;
  }
}
class Nil extends List {
  rich, poor: pot = 0;
  rich: List<poor>, 0 copy(0) {
    return this;
  } }

```

```

class Cons extends List {
  rich: pot = 1;
  poor: pot = 0;
  rich, poor: int elem;
  rich: List<rich,rich> next;
  poor: List<poor,poor> next;

  rich: List<poor>, 0 copy(0) {
    let res = new Cons in
    let _ = res.elem ← this.elem in
    let rnext = this.next.copy() in
    let _ = res.next ← rnext in
    return res;
  } }

```

- Space consumption of $l.copy()$: $n = \text{length}(l)$.
- The **potential** of a list l is $\text{length}(l)$ under the view **rich** or 0 under the view **poor**.

Copy example in RAJA

```

views rich, poor
class List {
  rich, poor: pot = 0;
  rich: List<poor>, 0 copy(0) {
    return null;
  }
}
class Nil extends List {
  rich, poor: pot = 0;
  rich: List<poor>, 0 copy(0) {
    return this;
  } }

```

```

class Cons extends List {
  rich: pot = 1;
  poor: pot = 0;
  rich, poor: int elem;
  rich: List<rich,rich> next;
  poor: List<poor,poor> next;

  rich: List<poor>, 0 copy(0) {
    let res = new Cons in
    let _ = res.elem ← this.elem in
    let rnext = this.next.copy() in
    let _ = res.next ← rnext in
    return res;
  } }

```

- Space consumption of $l.copy() : n = \text{length}(l)$.
- The **potential** of a list l is $\text{length}(l)$ under the view **rich** or 0 under the view **poor**.
- We pay the copy of $l : \text{List}^{\text{rich}}$ from its potential but we *cannot* copy $l : \text{List}^{\text{poor}}$.

Sketch of RAJA System

- RAJA program $P = \textit{annotated}$ FJEU program.
 - ① Set of *views* \mathcal{V} .
 - ② For each class C and view \mathbf{r} we have an annotated version $C^{\mathbf{r}}$.
 - ③ Potential:
 - $\diamond(C^{\mathbf{r}}) : \text{Class} \times \text{View} \rightarrow \mathbb{Q}^+$
 - ④ (Get- and set-) views for attributes:
 - $A^{\text{get}}(C^{\mathbf{r}}, \mathbf{a}) : \text{Class} \times \text{View} \times \text{Field} \rightarrow \text{View}$ (*get-view*)
 - $A^{\text{set}}(C^{\mathbf{r}}, \mathbf{a}) : \text{Class} \times \text{View} \times \text{Field} \rightarrow \text{View}$ (*set-view*)
 - ⑤ RAJA types for methods:
 - $M(C^{\mathbf{r}}, m) : \text{Class} \times \text{View} \times \text{Method} \rightarrow$
 $\mathcal{P}(\text{Views of Arguments} \rightarrow \text{Effect} \times \text{View of Result})$
 $(\mathbf{r}_1, \dots, \mathbf{r}_j \xrightarrow{m/m'} \mathbf{r}_0)$
 - Effect: pair of numbers m, m' representing potential consumed and released by the method.

Sketch of RAJA System

- RAJA program $P = \textit{annotated}$ FJEU program.
 - ① Set of *views* \mathcal{V} .
 - ② For each class C and view r we have an annotated version C^r .
 - ③ Potential:
 - $\diamond(C^r) : \text{Class} \times \text{View} \rightarrow \mathbb{Q}^+$
 - ④ (Get- and set-) views for attributes:
 - $A^{\text{get}}(C^r, a) : \text{Class} \times \text{View} \times \text{Field} \rightarrow \text{View}$ (*get-view*)
 - $A^{\text{set}}(C^r, a) : \text{Class} \times \text{View} \times \text{Field} \rightarrow \text{View}$ (*set-view*)
 - ⑤ RAJA types for methods:
 - $M(C^r, m) : \text{Class} \times \text{View} \times \text{Method} \rightarrow$
 $\mathcal{P}(\text{Views of Arguments} \rightarrow \text{Effect} \times \text{View of Result})$
 $(r_1, \dots, r_j \xrightarrow{m/m'} r_0)$
 - Effect: pair of numbers m, m' representing potential consumed and released by the method.

Sketch of RAJA System

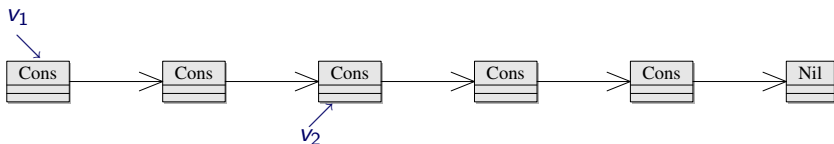
- RAJA program $P = \textit{annotated}$ FJEU program.
 - ① Set of *views* \mathcal{V} .
 - ② For each class C and view r we have an annotated version C^r .
 - ③ Potential:
 - $\diamond(C^r) : \text{Class} \times \text{View} \rightarrow \mathbb{Q}^+$
 - ④ (Get- and set-) views for attributes:
 - $A^{\text{get}}(C^r, a) : \text{Class} \times \text{View} \times \text{Field} \rightarrow \text{View}$ (**get-view**)
 - $A^{\text{set}}(C^r, a) : \text{Class} \times \text{View} \times \text{Field} \rightarrow \text{View}$ (**set-view**)
 - ⑤ RAJA types for methods:
 - $M(C^r, m) : \text{Class} \times \text{View} \times \text{Method} \rightarrow$
 $\mathcal{P}(\text{Views of Arguments} \rightarrow \text{Effect} \times \text{View of Result})$
 $(r_1, \dots, r_j \xrightarrow{m/m'} r_0)$
 - Effect: pair of numbers m, m' representing potential consumed and released by the method.

Sketch of RAJA System

- RAJA program $P = \textit{annotated}$ FJEU program.
 - ① Set of *views* \mathcal{V} .
 - ② For each class C and view \mathbf{r} we have an annotated version $C^{\mathbf{r}}$.
 - ③ Potential:
 - $\diamond(C^{\mathbf{r}}) : \text{Class} \times \text{View} \rightarrow \mathbb{Q}^+$
 - ④ (Get- and set-) views for attributes:
 - $A^{\text{get}}(C^{\mathbf{r}}, \mathbf{a}) : \text{Class} \times \text{View} \times \text{Field} \rightarrow \text{View}$ (**get**-view)
 - $A^{\text{set}}(C^{\mathbf{r}}, \mathbf{a}) : \text{Class} \times \text{View} \times \text{Field} \rightarrow \text{View}$ (**set**-view)
 - ⑤ RAJA types for methods:
 - $M(C^{\mathbf{r}}, m) : \text{Class} \times \text{View} \times \text{Method} \rightarrow$
 $\mathcal{P}(\text{Views of Arguments} \rightarrow \text{Effect} \times \text{View of Result})$
 $(\mathbf{r}_1, \dots, \mathbf{r}_j \xrightarrow{m/m'} \mathbf{r}_0)$
 - Effect: pair of numbers m, m' representing potential consumed and released by the method.

Potential

- Let $\Gamma = l_1 : \text{List}^{\text{rich}}, l_2 : \text{List}^{\text{poor}}$ and $\eta = [l_1 \mapsto v_1, l_2 \mapsto v_2]$ and σ be the following heap:



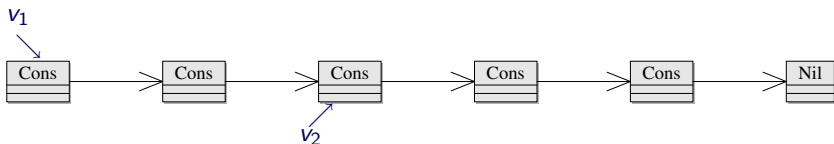
$$\text{pot}_\sigma(v_1 : r) = \begin{cases} 5 & r = \text{rich} \\ 0 & r = \text{poor} \end{cases}$$

$$\text{pot}_\sigma(v_2 : r) = \begin{cases} 3 & r = \text{rich} \\ 0 & r = \text{poor} \end{cases}$$

$$\text{pot}_\sigma(\eta : \Gamma) = \underbrace{\text{pot}_\sigma(v_1 : \text{rich})}_5 + \underbrace{\text{pot}_\sigma(v_2 : \text{poor})}_0 = 5$$

Potential

- Let $\Gamma = l_1 : \text{List}^{\text{rich}}, l_2 : \text{List}^{\text{poor}}$ and $\eta = [l_1 \mapsto v_1, l_2 \mapsto v_2]$ and σ be the following heap:



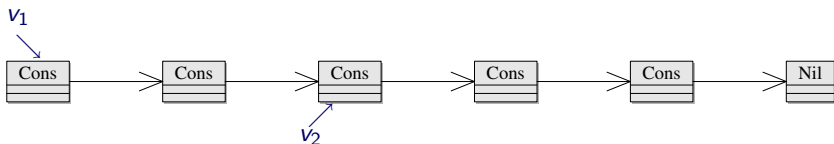
$$\text{pot}_\sigma(v_1 : r) = \begin{cases} 5 & r = \text{rich} \\ 0 & r = \text{poor} \end{cases}$$

$$\text{pot}_\sigma(v_2 : r) = \begin{cases} 3 & r = \text{rich} \\ 0 & r = \text{poor} \end{cases}$$

$$\text{pot}_\sigma(\eta : \Gamma) = \underbrace{\text{pot}_\sigma(v_1 : \text{rich})}_5 + \underbrace{\text{pot}_\sigma(v_2 : \text{poor})}_0 = 5$$

Potential

- Let $\Gamma = l_1 : \text{List}^{\text{rich}}, l_2 : \text{List}^{\text{poor}}$ and $\eta = [l_1 \mapsto v_1, l_2 \mapsto v_2]$ and σ be the following heap:



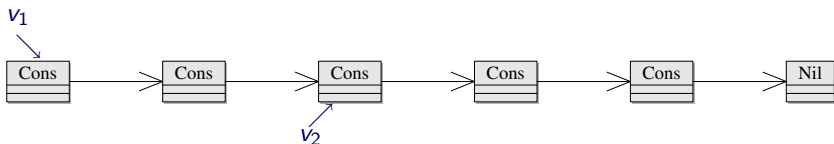
$$\text{pot}_\sigma(v_1 : \mathbf{r}) = \begin{cases} 5 & \mathbf{r} = \text{rich} \\ 0 & \mathbf{r} = \text{poor} \end{cases}$$

$$\text{pot}_\sigma(v_2 : \mathbf{r}) = \begin{cases} 3 & \mathbf{r} = \text{rich} \\ 0 & \mathbf{r} = \text{poor} \end{cases}$$

$$\text{pot}_\sigma(\eta : \Gamma) = \underbrace{\text{pot}_\sigma(v_1 : \text{rich})}_5 + \underbrace{\text{pot}_\sigma(v_2 : \text{poor})}_0 = 5$$

Potential

- Let $\Gamma = l_1 : \text{List}^{\text{rich}}, l_2 : \text{List}^{\text{poor}}$ and $\eta = [l_1 \mapsto v_1, l_2 \mapsto v_2]$ and σ be the following heap:



$$\text{pot}_\sigma(v_1 : r) = \begin{cases} 5 & r = \text{rich} \\ 0 & r = \text{poor} \end{cases}$$

$$\text{pot}_\sigma(v_2 : r) = \begin{cases} 3 & r = \text{rich} \\ 0 & r = \text{poor} \end{cases}$$

$$\text{pot}_\sigma(\eta : \Gamma) = \underbrace{\text{pot}_\sigma(v_1 : \text{rich})}_5 + \underbrace{\text{pot}_\sigma(v_2 : \text{poor})}_0 = 5$$

Main result

- $\eta, \sigma \vdash e \rightsquigarrow v, \tau$ means:
 - expression e evaluates successfully to value v
 - beginning with stack η and heap σ
 - and ending with heap τ
 - (with an unbounded free-list).
- We define a typing judgment $\Gamma \frac{n}{n'} e : C^r$ so that:
 - if $\Gamma \frac{n}{n'} e : C^r$ and $\eta, \sigma \vdash e \rightsquigarrow v, \tau$ then

e evaluates  if $|\text{free-list}| \geq n + \text{pot}_\sigma(\eta : \Gamma)$

Some RAJA typing rules

$$\frac{}{\emptyset \vdash \frac{1 + \diamond(C^r)}{0} \text{ new } C : C^r} (\diamond\text{New})$$

Some RAJA typing rules

$$\frac{}{\emptyset \vdash \frac{1 + \diamond(C^r)}{0} \text{ new } C : C^r} (\diamond\text{New})$$

$$\frac{}{x : C^r \vdash \frac{0}{1 + \diamond(C^r)} \text{ free } (x) : E^s} (\diamond\text{Free})$$

Some RAJA typing rules

$$\frac{}{\emptyset \vdash \frac{1 + \diamond(C^r)}{0} \text{ new } C : C^r} (\diamond\text{New})$$

$$\frac{}{x : C^r \vdash \frac{0}{1 + \diamond(C^r)} \text{ free } (x) : E^s} (\diamond\text{Free})$$

$$\frac{(E_1^{q_1}, \dots, E_j^{q_j} \xrightarrow{n/n'} E_0^{q_0}) \in M(C^r, m)}{x : C^r, y_1 : E_1^{q_1}, \dots, y_j : E_j^{q_j} \vdash \frac{n}{n'} x.m(y_1, \dots, y_j) : E_0^{q_0}} (\diamond\text{Invocation})$$

Sharing relation

- $\forall(C^r | C^{s_1}, \dots, C^{s_i})$: coinductively defined relation with:
for all $D < C : \diamond(D^r) \geq \diamond(D^{s_1}) + \dots + \diamond(D^{s_i})$, etc.

Sharing relation

- $\Downarrow(C^r \mid C^{s_1}, \dots, C^{s_i})$: coinductively defined relation with:
for all $D <: C : \diamond(D^r) \geq \diamond(D^{s_1}) + \dots + \diamond(D^{s_i})$, etc.
- $\Downarrow(\text{List}^{\text{rich}} \mid \text{List}^{\text{rich}}, \text{List}^{\text{poor}})$ ✓

Sharing relation

- $\Downarrow(C^r \mid C^{s_1}, \dots, C^{s_i})$: coinductively defined relation with:
 for all $D <: C : \diamond(D^r) \geq \diamond(D^{s_1}) + \dots + \diamond(D^{s_i})$, etc.
- $\Downarrow(\text{List}^{\text{rich}} \mid \text{List}^{\text{rich}}, \text{List}^{\text{poor}})$ ✓ $\Downarrow(\text{List}^{\text{rich}} \mid \text{List}^{\text{rich}}, \text{List}^{\text{rich}})$ ✗

Sharing relation

- If this were allowed ...

$$l : \text{List}^{\text{rich}} \stackrel{0}{\mid} \stackrel{0}{\text{let } _ = l.\text{copy}() \text{ in } l.\text{copy}()} : \text{List}^{\text{poor}}$$

Sharing relation

- If this were allowed ...

$$l : \text{List}^{\text{rich}} \stackrel{0}{\vdash} \text{let } _ = l.\text{copy}() \text{ in } l.\text{copy}() : \text{List}^{\text{poor}}$$

- ... heap consumption would be $2|l|$, but prediction would be $\text{pot}_\sigma(l : \text{rich}) = |l| \Rightarrow$ unsound!

Sharing relation

- If this were allowed ...

$$l : \text{List}^{\text{rich}} \vdash_0^0 \text{ let } _ = l.\text{copy}() \text{ in } l.\text{copy}() : \text{List}^{\text{poor}}$$

- ... heap consumption would be $2|l|$, but prediction would be $\text{pot}_\sigma(l : \text{rich}) = |l|$. \Rightarrow unsound!
- The rule (\diamond Share) enables multiple (sound) uses of a variable.

$$\frac{\Gamma, y_1 : D^{\mathfrak{q}_1}, y_2 : D^{\mathfrak{q}_2} \vdash_{n'}^n e : C^r \quad \forall (D^s \mid D^{\mathfrak{q}_1}, D^{\mathfrak{q}_2})}{\Gamma, x : D^s \vdash_{n'}^n e[x/y_1, x/y_2] : C^r} (\diamond \text{Share})$$

Sharing relation

- If this were allowed ...

$$l : \text{List}^{\text{rich}} \stackrel{0}{|} \text{ let } _ = l.\text{copy}() \text{ in } l.\text{copy}() : \text{List}^{\text{poor}}$$

- ... heap consumption would be $2|l|$, but prediction would be $\text{pot}_\sigma(l : \text{rich}) = |l|$. \Rightarrow unsound!
- The rule ($\diamond\text{Share}$) enables multiple (**sound**) uses of a variable.

$$\frac{\Gamma, y_1 : D^{q_1}, y_2 : D^{q_2} \stackrel{n}{|}_{n'} e : C^r \quad \forall (D^s | D^{q_1}, D^{q_2})}{\Gamma, x : D^s \stackrel{n}{|}_{n'} e[x/y_1, x/y_2] : C^r} (\diamond\text{Share})$$

- ($\diamond\text{Share}$) is a problematic rule for the implementation because
 - It is non-syntax-directed.
 - The intermediate views q_1, q_2 need to be found out.

Sharing relation

- If this were allowed ...

$$l : \text{List}^{\text{rich}} \stackrel{0}{|} \text{ let } _ = l.\text{copy}() \text{ in } l.\text{copy}() : \text{List}^{\text{poor}}$$

- ... heap consumption would be $2|l|$, but prediction would be $\text{pot}_\sigma(l : \text{rich}) = |l|$. \Rightarrow unsound!
- The rule ($\diamond\text{Share}$) enables multiple (**sound**) uses of a variable.

$$\frac{\Gamma, y_1 : D^{q_1}, y_2 : D^{q_2} \stackrel{n}{|}_{n'} e : C^r \quad \forall (D^s | D^{q_1}, D^{q_2})}{\Gamma, x : D^s \stackrel{n}{|}_{n'} e[x/y_1, x/y_2] : C^r} (\diamond\text{Share})$$

- ($\diamond\text{Share}$) is a problematic rule for the implementation because
 - It is non-syntax-directed.
 - The intermediate views q_1, q_2 need to be found out.

Sharing relation

- If this were allowed ...

$$l : \text{List}^{\text{rich}} \stackrel{0}{|} \text{ let } _ = l.\text{copy}() \text{ in } l.\text{copy}() : \text{List}^{\text{poor}}$$

- ... heap consumption would be $2|l|$, but prediction would be $\text{pot}_\sigma(l : \text{rich}) = |l|$. \Rightarrow unsound!
- The rule ($\diamond\text{Share}$) enables multiple (**sound**) uses of a variable.

$$\frac{\Gamma, y_1 : D^{q_1}, y_2 : D^{q_2} \stackrel{n}{|}_{n'} e : C^r \quad \forall (D^s | D^{q_1}, D^{q_2})}{\Gamma, x : D^s \stackrel{n}{|}_{n'} e[x/y_1, x/y_2] : C^r} (\diamond\text{Share})$$

- ($\diamond\text{Share}$) is a problematic rule for the implementation because
 - It is non-syntax-directed.
 - The intermediate views q_1, q_2 need to be found out.

Gaining potential from this

- Method body rule (*too weak*)

$$\frac{\forall(C^r \mid C^q, C^s) \quad \text{this}: C^r, x_1: E_1^{r_1}, \dots, x_j: E_j^{r_j} \vdash_{n/n'} M_{\text{body}}(C, m) : E_0^{r_0}}{\vdash m : E_1^{r_1}, \dots, E_j^{r_j} \xrightarrow{n/n'} E_0^{r_0} \in M(C^r, m) \text{ ok}} \quad (\diamond MBody)$$

- Example:

```
class Cons extends List {
  ..
  rich: List<poor>, 0 copy() {
    Cons<poor> res = new Cons;
    res.elem = elem;
    res.next = this.next.copy();
    return res;
  }
}
```

- where $\forall(C^{\text{rich}} \mid \text{Cons}^1, \text{Cons}^{\text{rich}-1})$
- Intermediate views q, s are automatically inferred using **algorithmic views** $r \dot{-} n$ and n .

Gaining potential from this

- Method body rule (*too weak*)

$$\frac{\forall(C^r | C^q, C^s) \quad \text{this}: C^r, x_1: E_1^{r_1}, \dots, x_j: E_j^{r_j} \vdash_{n/n'} M_{\text{body}}(C, m) : E_0^{r_0}}{\vdash m : E_1^{r_1}, \dots, E_j^{r_j} \xrightarrow{n/n'} E_0^{r_0} \in M(C^r, m) \text{ ok}} \quad (\diamond MBody)$$

- Example:

```
class Cons extends List {
  ..
  rich: List<poor>, 0 copy(0) {
    Cons<poor> res = new Cons;
    res.elem = elem;
    res.next = this.next.copy();
    return res;
  }
}
```

- where $\forall(C^{\text{rich}} | C^1, C^{\text{rich} - 1})$
- Intermediate views q, s are automatically inferred using **algorithmic views** $r \dot{-} n$ and n .

Gaining potential from this

- Method body rule (*too weak*)

$$\frac{\forall(C^r \mid C^q, C^s) \quad \text{this}: C^r, x_1: E_1^{r_1}, \dots, x_j: E_j^{r_j} \vdash_{n/n'} M_{\text{body}}(C, m) : E_0^{r_0}}{\vdash m : E_1^{r_1}, \dots, E_j^{r_j} \xrightarrow{n/n'} E_0^{r_0} \in M(C^r, m) \text{ ok}} \quad (\diamond \text{MBody})$$

- Example:

```
class Cons extends List {
    ...
    rich: List<poor>, 0 copy(0) { /* this : Consrich | 0 Mbody(Cons, copy) : Listpoor */
        Cons<poor> res = new Cons;
        res.elem = elem;
        res.next = this.next.copy();
        return res;
    }
}
```

- where $\forall(C^{\text{rich}} \mid C^1, C^{\text{rich}-1})$
- Intermediate views q, s are automatically inferred using **algorithmic views** $r \dot{-} n$ and n .

Gaining potential from this

- Method body rule (*too weak*)

$$\frac{\forall(C^r \mid C^q, C^s) \quad \text{this}: C^r, x_1: E_1^{r_1}, \dots, x_j: E_j^{r_j} \vdash_{n/n'} M_{\text{body}}(C, m) : E_0^{r_0}}{\vdash m : E_1^{r_1}, \dots, E_j^{r_j} \xrightarrow{n/n'} E_0^{r_0} \in M(C^r, m) \text{ ok}} \quad (\diamond \text{MBody})$$

- Example:

```
class Cons extends List {
  ...
  rich: List<poor>, 0 copy(0) { /* this : Consrich | 0 Mbody(Cons, copy) : Listpoor */
    Cons<poor> res = new Cons; /* potential available: 0 */
    res.elem = elem;
    res.next = this.next.copy();
    return res;
  }
}
```

- where $\forall(C^{\text{rich}} \mid \text{Cons}^1, \text{Cons}^{\text{rich}-1})$
- Intermediate views q, s are automatically inferred using **algorithmic views** $r \dot{-} n$ and n .

Gaining potential from this

- Method body rule (*too weak*)

$$\frac{\forall(C^r \mid C^q, C^s) \quad \text{this}: C^r, x_1: E_1^{r_1}, \dots, x_j: E_j^{r_j} \vdash_{n/n'} M_{\text{body}}(C, m) : E_0^{r_0}}{\vdash m : E_1^{r_1}, \dots, E_j^{r_j} \xrightarrow{n/n'} E_0^{r_0} \in M(C^r, m) \text{ ok}} \quad (\diamond MBody)$$

- Example:

```
class Cons extends List {
  ...
  rich: List<poor>, 0 copy(0) { /* this : Consrich | 0 Mbody(Cons, copy) : Listpoor */
    Cons<poor> res = new Cons; /* potential available: 0 ✗ */
    res.elem = elem;
    res.next = this.next.copy();
    return res;
  }
}
```

- where $\forall(C^{\text{rich}} \mid \text{Cons}^1, \text{Cons}^{\text{rich}-1})$
- Intermediate views q, s are automatically inferred using **algorithmic views** $r \dot{-} n$ and n .

Gaining potential from this

- Method body rule

$$\frac{\forall(C^r \mid C^q, C^s) \quad \text{this}: C^q, x_1: E_1^{r_1}, \dots, x_j: E_j^{r_j} \mid \frac{n + \diamond(C^s)}{n'} \quad M_{\text{body}}(C, m): E_0^{r_0}}{\vdash m: E_1^{r_1}, \dots, E_j^{r_j} \xrightarrow{n/n'} E_0^{r_0} \in M(C^r, m) \text{ ok}} \quad (\diamond \text{MBody})$$

- Example:

```
class Cons extends List {
    ...
    rich: List<poor>, 0 copy(0) {
        Cons<poor> res = new Cons;
        res.elem = elem;
        res.next = this.next.copy();
        return res;
    }
}
```

- where $\forall(\text{Cons}^{\text{rich}} \mid \text{Cons}^1, \text{Cons}^{\text{rich}-1})$
- Intermediate views q, s are automatically inferred using **algorithmic views** $r \dot{-} n$ and n .

Gaining potential from this

- Method body rule

$$\frac{\forall(C^r \mid C^q, C^s) \quad \text{this}: C^q, x_1: E_1^{r_1}, \dots, x_j: E_j^{r_j} \mid \frac{n + \diamond(C^s)}{n'} \quad M_{\text{body}}(C, m) : E_0^{r_0}}{\vdash m : E_1^{r_1}, \dots, E_j^{r_j} \xrightarrow{n/n'} E_0^{r_0} \in M(C^r, m) \text{ ok}} \quad (\diamond \text{MBody})$$

- Example:

```
class Cons extends List {
    ...
    rich: List<poor>, 0 copy(0) {
        Cons<poor> res = new Cons;
        res.elem = elem;
        res.next = this.next.copy();
        return res;
    }
}
```

- where $\forall(C^{\text{rich}} \mid \text{Cons}^1, \text{Cons}^{\text{rich}-1})$
- Intermediate views q, s are automatically inferred using **algorithmic views** $r \dot{-} n$ and n .

Gaining potential from this

- Method body rule

$$\frac{\forall(C^r | C^q, C^s) \quad \text{this}: C^q, x_1: E_1^{r_1}, \dots, x_j: E_j^{r_j} \mid \frac{n + \diamond(C^s)}{n'} \quad M_{\text{body}}(C, m): E_0^{r_0}}{\vdash m: E_1^{r_1}, \dots, E_j^{r_j} \xrightarrow{n/n'} E_0^{r_0} \in M(C^r, m) \text{ ok}} \quad (\diamond MBody)$$

- Example:

```
class Cons extends List {
    rich: List<poor>, 0 copy(0) {
        Cons<poor> res = new Cons;
        res.elem = elem;
        res.next = this.next.copy();
        return res;
    }
}
```

- where $\forall(C^{\text{rich}} | C^{\text{poor}}, C^{\text{rich}-1})$
- Intermediate views q, s are automatically inferred using **algorithmic views** $r \dot{-} n$ and n .

Gaining potential from this

- Method body rule

$$\frac{\forall(C^r \mid C^q, C^s) \quad \text{this}: C^q, x_1: E_1^{r_1}, \dots, x_j: E_j^{r_j} \mid \frac{n + \diamond(C^s)}{n'} \quad M_{\text{body}}(C, m): E_0^{r_0}}{\vdash m: E_1^{r_1}, \dots, E_j^{r_j} \xrightarrow{n/n'} E_0^{r_0} \in M(C^r, m) \text{ ok}} \quad (\diamond MBody)$$

- Example:

```
class Cons extends List {
  ...
  rich: List<poor>, 0 copy(0) { /* this: Consrich-1 |  $\frac{0 + \diamond(\text{Cons}^1)}{0}$  ... */
    Cons<poor> res = new Cons;
    res.elem = elem;
    res.next = this.next.copy();
    return res;
  }
}
```

- where $\forall(\text{Cons}^{\text{rich}} \mid \text{Cons}^1, \text{Cons}^{\text{rich}-1})$
- Intermediate views q, s are automatically inferred using **algorithmic views** $r \dot{-} n$ and n .

Gaining potential from this

- Method body rule

$$\frac{\forall(C^r | C^q, C^s) \quad \text{this}: C^q, x_1: E_1^{r_1}, \dots, x_j: E_j^{r_j} \mid \frac{n + \diamond(C^s)}{n'} \quad M_{\text{body}}(C, m) : E_0^{r_0}}{\vdash m : E_1^{r_1}, \dots, E_j^{r_j} \xrightarrow{n/n'} E_0^{r_0} \in M(C^r, m) \text{ ok}} \quad (\diamond MBody)$$

- Example:

```
class Cons extends List {
  ...
  rich: List<poor>, 0 copy(0) { /* this : Consrich-1 |  $\frac{0 + \diamond(\text{Cons}^1)}{0}$  ... */
    Cons<poor> res = new Cons; /* potential available: 1 */
    res.elem = elem;
    res.next = this.next.copy();
    return res;
  }
}
```

- where $\forall(\text{Cons}^{\text{rich}} | \text{Cons}^1, \text{Cons}^{\text{rich}-1})$
- Intermediate views q, s are automatically inferred using **algorithmic views** $r \dot{-} n$ and n .

Gaining potential from this

- Method body rule

$$\frac{\forall(C^r | C^q, C^s) \quad \text{this}: C^q, x_1: E_1^{r_1}, \dots, x_j: E_j^{r_j} \mid \frac{n + \diamond(C^s)}{n'} \quad M_{\text{body}}(C, m) : E_0^{r_0}}{\vdash m : E_1^{r_1}, \dots, E_j^{r_j} \xrightarrow{n/n'} E_0^{r_0} \in M(C^r, m) \text{ ok}} \quad (\diamond MBody)$$

- Example:

```
class Cons extends List {
  ...
  rich: List<poor>, 0 copy(0) { /* this : Consrich-1 |  $\frac{0 + \diamond(\text{Cons}^1)}{0}$  ... */
    Cons<poor> res = new Cons; /* potential available: 1 */
    res.elem = elem;
    res.next = this.next.copy();
    return res;
  }
}
```

- where $\forall(\text{Cons}^{\text{rich}} \mid \text{Cons}^1, \text{Cons}^{\text{rich}-1})$
- Intermediate views q, s are automatically inferred using **algorithmic views** $r \dot{-} n$ and n .

Gaining potential from `this`

- Method body rule

$$\frac{\forall(C^r | C^q, C^s) \quad \text{this}: C^q, x_1: E_1^{r_1}, \dots, x_j: E_j^{r_j} \mid \frac{n + \diamond(C^s)}{n'} \quad M_{\text{body}}(C, m) : E_0^{r_0}}{\vdash m : E_1^{r_1}, \dots, E_j^{r_j} \xrightarrow{n/n'} E_0^{r_0} \in M(C^r, m) \text{ ok}} \quad (\diamond MBody)$$

- Example:

```
class Cons extends List {
  ...
  rich: List<poor>, 0 copy(0) { /* this : Consrich-1 |  $\frac{0 + \diamond(\text{Cons}^1)}{0}$  ... */
    Cons<poor> res = new Cons; /* potential available: 1 */
    res.elem = elem;
    res.next = this.next.copy();
    return res;
  }
}
```

- where $\forall(\text{Cons}^{\text{rich}} \mid \text{Cons}^1, \text{Cons}^{\text{rich}-1})$
- Intermediate views \mathbf{q}, \mathbf{s} are automatically inferred using **algorithmic views** $\mathbf{r} \dot{-} \mathbf{n}$ and \mathbf{n} .

Algorithmic views

Let $\mathbf{r}, \mathbf{s} \in \mathcal{V}, n \in \mathbb{N}$.

- $\mathbf{r} \dot{-} \mathbf{n} \Rightarrow \mathbf{r} \dot{-} \mathbf{n}$ is defined like \mathbf{r} but with
 $\Rightarrow \diamond(C^{\mathbf{r} \dot{-} \mathbf{n}}) = \diamond(C^{\mathbf{r}}) \dot{-} n$.
- $\mathbf{n} \Rightarrow \diamond(C^{\mathbf{n}}) = n$.
- $\mathbf{r} + \mathbf{s} \Rightarrow C^{\mathbf{r} + \mathbf{s}}$ is a formal sum operation of $C^{\mathbf{r}}$ and $C^{\mathbf{s}}$,
 $\Rightarrow \diamond(C^{\mathbf{r} + \mathbf{s}}) = \diamond(C^{\mathbf{r}}) + \diamond(C^{\mathbf{s}})$
- $\forall (C^{\mathbf{r}} | C^{\mathbf{n}}, C^{\mathbf{r} - \mathbf{n}}) \quad \forall (C^{\mathbf{r}} | C^{2\mathbf{n}}, C^{2\mathbf{s}}) \iff C^{\mathbf{r}} \prec C^{\mathbf{n} + \mathbf{s}} \quad C^{\mathbf{r} + \mathbf{0}} = C^{\mathbf{r}}$

Algorithmic views

Let $\mathbf{r}, \mathbf{s} \in \mathcal{V}, n \in \mathbb{N}$.

- $\mathbf{r} \dot{-} \mathbf{n}$ \Rightarrow $\mathbf{r} \dot{-} \mathbf{n}$ is defined like \mathbf{r} but with
 $\Rightarrow \diamond(C^{\mathbf{r} \dot{-} \mathbf{n}}) = \diamond(C^{\mathbf{r}}) \dot{-} n$.
- \mathbf{n} $\Rightarrow \diamond(C^{\mathbf{n}}) = n$.
- $\mathbf{r} + \mathbf{s}$ $\Rightarrow C^{\mathbf{r} + \mathbf{s}}$ is a formal sum operation of $C^{\mathbf{r}}$ and $C^{\mathbf{s}}$,
 $\Rightarrow \diamond(C^{\mathbf{r} + \mathbf{s}}) = \diamond(C^{\mathbf{r}}) + \diamond(C^{\mathbf{s}})$
- $\forall (C^{\mathbf{r}} | C^{\mathbf{n}}, C^{\mathbf{r} - \mathbf{n}})$ $\forall (C^{\mathbf{r}} | C^{2\mathbf{n}}, C^{2\mathbf{s}}) \Leftrightarrow C^{\mathbf{r}} \leq C^{\mathbf{n} + \mathbf{s}}$ $C^{\mathbf{r} + \mathbf{0}} = C^{\mathbf{r}}$

Algorithmic views

Let $\mathbf{r}, \mathbf{s} \in \mathcal{V}, n \in \mathbb{N}$.

- $\mathbf{r} \dot{-} \mathbf{n} \Rightarrow \mathbf{r} \dot{-} \mathbf{n}$ is defined like \mathbf{r} but with
 $\Rightarrow \diamond(C^{\mathbf{r} \dot{-} \mathbf{n}}) = \diamond(C^{\mathbf{r}}) \dot{-} n$.

- $\mathbf{n} \Rightarrow \diamond(C^{\mathbf{n}}) = n$.

- $\mathbf{r} + \mathbf{s} \Rightarrow C^{\mathbf{r} + \mathbf{s}}$ is a formal sum operation of $C^{\mathbf{r}}$ and $C^{\mathbf{s}}$,
 $\Rightarrow \diamond(C^{\mathbf{r} + \mathbf{s}}) = \diamond(C^{\mathbf{r}}) + \diamond(C^{\mathbf{s}})$

- $\forall (C^{\mathbf{r}} | C^{\mathbf{n}}, C^{\mathbf{r}-\mathbf{n}}) \quad \forall (C^{\mathbf{r}} | C^{\mathbf{s}}, C^{\mathbf{s}}) \Leftrightarrow C^{\mathbf{r}} \leq C^{\mathbf{r} + \mathbf{s}} \quad C^{\mathbf{r} + \mathbf{0}} = C^{\mathbf{r}}$

Algorithmic views

Let $\mathbf{r}, \mathbf{s} \in \mathcal{V}, n \in \mathbb{N}$.

- \bullet $\mathbf{r} \dot{-} \mathbf{n} \Rightarrow \mathbf{r} \dot{-} \mathbf{n}$ is defined like \mathbf{r} but with
 $\Rightarrow \diamond(C^{\mathbf{r} \dot{-} \mathbf{n}}) = \diamond(C^{\mathbf{r}}) \dot{-} n$.
- \bullet $\mathbf{n} \Rightarrow \diamond(C^{\mathbf{n}}) = n$.
- \bullet $\mathbf{r} + \mathbf{s} \Rightarrow C^{\mathbf{r} + \mathbf{s}}$ is a formal sum operation of $C^{\mathbf{r}}$ and $C^{\mathbf{s}}$,
 $\Rightarrow \diamond(C^{\mathbf{r} + \mathbf{s}}) = \diamond(C^{\mathbf{r}}) + \diamond(C^{\mathbf{s}})$
- \bullet $\forall(C^{\mathbf{r}} | C^{\mathbf{n}}, C^{\mathbf{r} - \mathbf{n}}) \quad \forall(C^{\mathbf{r}} | C^{\mathbf{s}_1}, C^{\mathbf{s}_2}) \iff C^{\mathbf{r}} <: C^{\mathbf{s}_1 + \mathbf{s}_2} \quad C^{\mathbf{r} + \mathbf{0}} = C^{\mathbf{r}}$

Algorithmic views

Let $\mathbf{r}, \mathbf{s} \in \mathcal{V}, n \in \mathbb{N}$.

- \bullet $\mathbf{r} \dot{-} \mathbf{n} \Rightarrow \mathbf{r} \dot{-} \mathbf{n}$ is defined like \mathbf{r} but with
 $\Rightarrow \diamond(C^{\mathbf{r} \dot{-} \mathbf{n}}) = \diamond(C^{\mathbf{r}}) \dot{-} n$.
- \bullet $\mathbf{n} \Rightarrow \diamond(C^{\mathbf{n}}) = n$.
- \bullet $\mathbf{r} + \mathbf{s} \Rightarrow C^{\mathbf{r} + \mathbf{s}}$ is a formal sum operation of $C^{\mathbf{r}}$ and $C^{\mathbf{s}}$,
 $\Rightarrow \diamond(C^{\mathbf{r} + \mathbf{s}}) = \diamond(C^{\mathbf{r}}) + \diamond(C^{\mathbf{s}})$
- \bullet $\forall(C^{\mathbf{r}} | C^{\mathbf{n}}, C^{\mathbf{r} - \mathbf{n}}) \quad \forall(C^{\mathbf{r}} | C^{\mathbf{s}_1}, C^{\mathbf{s}_2}) \iff C^{\mathbf{r}} <: C^{\mathbf{s}_1 + \mathbf{s}_2} \quad C^{\mathbf{r} + \mathbf{0}} = C^{\mathbf{r}}$

Algorithmic views

Let $\mathbf{r}, \mathbf{s} \in \mathcal{V}, n \in \mathbb{N}$.

- $\mathbf{r} \dot{-} \mathbf{n} \Rightarrow \mathbf{r} \dot{-} \mathbf{n}$ is defined like \mathbf{r} but with
 $\Rightarrow \diamond(C^{\mathbf{r} \dot{-} \mathbf{n}}) = \diamond(C^{\mathbf{r}}) \dot{-} n$.
- $\mathbf{n} \Rightarrow \diamond(C^{\mathbf{n}}) = n$.
- $\mathbf{r} + \mathbf{s} \Rightarrow C^{\mathbf{r} + \mathbf{s}}$ is a formal sum operation of $C^{\mathbf{r}}$ and $C^{\mathbf{s}}$,
 $\Rightarrow \diamond(C^{\mathbf{r} + \mathbf{s}}) = \diamond(C^{\mathbf{r}}) + \diamond(C^{\mathbf{s}})$
- $\forall(C^{\mathbf{r}} | C^{\mathbf{n}}, C^{\mathbf{r} - \mathbf{n}}) \quad \forall(C^{\mathbf{r}} | C^{\mathbf{s}_1}, C^{\mathbf{s}_2}) \iff C^{\mathbf{r}} <: C^{\mathbf{s}_1 + \mathbf{s}_2} \quad C^{\mathbf{r} + \mathbf{0}} = C^{\mathbf{r}}$

Idea of the algorithm

- Subtyping and sharing are integrated in the rules.
- Variable occurrences are **annotated** with a view (x^r).
- The views from the different occurrences are summed up.
- Judgment: $\Delta^\Psi \vdash_{n/n'} e^\circ \Leftarrow C^\gamma$
- where Δ is an FJEU context and Ψ a map from variables to *algorithmic* views.
- $\Rightarrow \Delta^\Psi$ is a map from variables to **algorithmic RAJA types**.

$$\text{typecheck}(\Delta, e^\circ, C^\gamma) = \left\{ \begin{array}{l} \text{true} \\ \text{false} \end{array} \right\}$$

Idea of the algorithm

- Subtyping and sharing are integrated in the rules.
- Variable occurrences are **annotated** with a view (x^r).
- The views from the different occurrences are summed up.
- Judgment: $\Delta^\Psi \vdash_{n/n'} e^\circ \Leftarrow C^\gamma$
- where Δ is an FJEU context and Ψ a map from variables to *algorithmic* views.
- $\Rightarrow \Delta^\Psi$ is a map from variables to **algorithmic RAJA types**.

$$\text{typecheck}(\Delta, e^\circ, C^\gamma) = \left\{ \begin{array}{l} \text{true} \\ \text{false} \end{array} \right\}$$

Idea of the algorithm

- Subtyping and sharing are integrated in the rules.
- Variable occurrences are **annotated** with a view (x^r).
- The views from the different occurrences are summed up.
- Judgment: $\Delta^\Psi \vdash_{n/n'} e^\circ \Leftarrow C^\gamma$
- where Δ is an FJEU context and Ψ a map from variables to *algorithmic* views.
- $\Rightarrow \Delta^\Psi$ is a map from variables to **algorithmic RAJA types**.

$$\text{typecheck}(\Delta, e^\circ, C^\gamma) = \left\{ \begin{array}{l} \text{true} \\ \text{false} \end{array} \right.$$

Idea of the algorithm

- Subtyping and sharing are integrated in the rules.
- Variable occurrences are **annotated** with a view (x^r).
- The views from the different occurrences are summed up.
- Judgment: $\Delta^\Psi \stackrel{n}{n'} e^\circ \Leftarrow C^\gamma$
- where Δ is an FJEU context and Ψ a map from variables to *algorithmic* views.
- $\Rightarrow \Delta^\Psi$ is a map from variables to **algorithmic RAJA types**.

$$\text{typecheck}(\Delta, e^\circ, C^\gamma) = \left\{ \begin{array}{l} \text{true} \\ \text{false} \end{array} \right.$$

Idea of the algorithm

- Subtyping and sharing are integrated in the rules.
- Variable occurrences are **annotated** with a view (x^r).
- The views from the different occurrences are summed up.
- Judgment: $\Delta^\Psi \stackrel{n}{n'} e^\circ \Leftarrow C^\gamma$
- where Δ is an FJEU context and Ψ a map from variables to *algorithmic* views.
- $\Rightarrow \Delta^\Psi$ is a map from variables to **algorithmic RAJA types**.

$$\text{typecheck}(\Delta, e^\circ, C^\gamma) = \begin{cases} (\Psi, n, n') & \text{if } \Delta^\Psi \stackrel{n}{n'} e^\circ \Leftarrow C^\gamma \\ \text{fail} & \text{otherwise} \end{cases}$$

Idea of the algorithm

- Subtyping and sharing are integrated in the rules.
- Variable occurrences are **annotated** with a view (x^r).
- The views from the different occurrences are summed up.
- Judgment: $\Delta^\Psi \stackrel{n}{n'} e^\circ \Leftarrow C^\gamma$
- where Δ is an FJEU context and Ψ a map from variables to *algorithmic* views.
- $\Rightarrow \Delta^\Psi$ is a map from variables to **algorithmic RAJA types**.

$$\text{typecheck}(\Delta, e^\circ, C^\gamma) = \begin{cases} (\Psi, n, n') & \text{if } \Delta^\Psi \stackrel{n}{n'} e^\circ \Leftarrow C^\gamma \\ \text{fail} & \text{otherwise} \end{cases}$$

Idea of the algorithm

- Subtyping and sharing are integrated in the rules.
- Variable occurrences are **annotated** with a view (x^r).
- The views from the different occurrences are summed up.
- Judgment: $\Delta^\Psi \stackrel{n}{n'} e^\circ \Leftarrow C^\gamma$
- where Δ is an FJEU context and Ψ a map from variables to *algorithmic* views.
- $\Rightarrow \Delta^\Psi$ is a map from variables to **algorithmic RAJA types**.

$$\text{typecheck}(\Delta, e^\circ, C^\gamma) = \begin{cases} (\Psi, n, n') & \text{if } \Delta^\Psi \stackrel{n}{n'} e^\circ \Leftarrow C^\gamma \\ \text{fail} & \text{otherwise} \end{cases}$$

Idea of the algorithm

- Subtyping and sharing are integrated in the rules.
- Variable occurrences are **annotated** with a view (x^r).
- The views from the different occurrences are summed up.
- Judgment: $\Delta^\Psi \vdash_{n/n'} e^\circ \Leftarrow C^\gamma$
- where Δ is an FJEU context and Ψ a map from variables to *algorithmic* views.
- $\Rightarrow \Delta^\Psi$ is a map from variables to **algorithmic RAJA types**.

$$\text{typecheck}(\Delta, e^\circ, C^\gamma) = \begin{cases} (\Psi, n, n') & \text{if } \Delta^\Psi \vdash_{n/n'} e^\circ \Leftarrow C^\gamma \\ \text{fail} & \text{otherwise} \end{cases}$$

Idea of the algorithm

- Subtyping and sharing are integrated in the rules.
- Variable occurrences are **annotated** with a view (x^r).
- The views from the different occurrences are summed up.
- Judgment: $\Delta^\Psi \stackrel{n}{n'} e^\circ \Leftarrow C^\gamma$
- where Δ is an FJEU context and Ψ a map from variables to *algorithmic* views.
- $\Rightarrow \Delta^\Psi$ is a map from variables to **algorithmic RAJA types**.

$$\text{typecheck}(\Delta, e^\circ, C^\gamma) = \begin{cases} (\Psi, n, n') & \text{if } \Delta^\Psi \stackrel{n}{n'} e^\circ \Leftarrow C^\gamma \\ \text{fail} & \text{otherwise} \end{cases}$$

Typechecking Algorithm on an Example

```
rich: List<poor>,0 copy(0) {  
  
  let res = new Cons in  
  
  let _ = res.elem ← this.elem in  
  
  let rnext = this.next.copy() in  
  
  let _ = res.next ← rnext in  
  
  return res; }
```

Typechecking Algorithm on an Example

```
rich: List<poor>,0 copy(0) {
  let res = new Cons in
  let _ = [res as 0].elem ← [this as 0].elem in
  let rnext = [this as rich-1].next.copy() in
  let _ = [res as poor].next ← [rnext as poor] in
  return [res as poor]; }
```

view annotations are provided by the user

Typechecking Algorithm on an Example

```
rich: List<poor>,0 copy(0) {
  let List res = new Cons in
    FJEU types annotations are automatically inferred
  let List _ = [res as 0].elem ← [this as 0].elem in
  let List rnext = [this as rich-1].next.copy() in
  let List _ = [res as poor].next ← [rnext as poor] in
  return [res as poor]; }
```

Typechecking Algorithm on an Example

$$\text{this} : \text{Cons}^? \stackrel{?}{\vdash} M_{\text{body}}(\text{Cons}, \text{copy}) \Leftarrow \text{List}^{\text{poor}}$$

```
rich: List<poor>,0 copy(0) {
  let List res = new Cons in
  let List _ = [res as 0].elem ← [this as 0].elem in
  let List rnext = [this as rich-1].next.copy() in
  let List _ = [res as poor].next ← [rnext as poor] in
  return [res as poor]; }
```

Typechecking Algorithm on an Example

$$\text{this} : \text{Cons}^? \vdash^? M_{\text{body}}(\text{Cons}, \text{copy}) \Leftarrow \text{List}^{\text{poor}}$$

```
rich: List<poor>,0 copy(0) {
  let List<poor+poor> res = new Cons in
  let List _ = [res as 0].elem ← [this as 0].elem in
  let List rnext = [this as rich-1].next.copy() in
  let List _ = [res as poor].next ← [rnext as poor] in
  return [res as poor]; }
```

Typechecking Algorithm on an Example

$$\text{this} : \text{Cons}^? \vdash^? M_{\text{body}}(\text{Cons}, \text{copy}) \Leftarrow \text{List}^{\text{poor}}$$

```
rich: List<poor>,0 copy(0) {
  let List<poor+poor> res = new Cons in
  let List _ = [res as 0].elem ← [this as 0].elem in
  let List<poor> rnext = [this as rich-1].next.copy() in
  let List _ = [res as poor].next ← [rnext as poor] in
  return [res as poor]; }
```

Typechecking Algorithm on an Example

$$\text{this} : \text{Cons}^{\text{rich}-1} \mid \frac{1 = 0 + \diamond(\text{Cons}^1)}{0} \quad M_{\text{body}}(\text{Cons}, \text{copy}) \Leftarrow \text{List}^{\text{poor}}$$

```
rich: List<poor>, 0 copy(0) {
```

```
let List<poor+poor> res = new Cons in
```

```
let List _ = [res as 0].elem ← [this as 0].elem in
```

```
let List<poor> rnext = [this as rich-1].next.copy() in
```

```
let List _ = [res as poor].next ← [rnext as poor] in
```

```
return [res as poor]; }
```

Correctness of the Algorithm

Lemma (Soundness of algorithmic RAJA typing)

If $\Delta^\Psi \vdash_{n'}^n e \Leftarrow C^\gamma$ then $\Delta^\Psi \vdash_{n'} |e| : C^\gamma$.

Lemma (Completeness of algorithmic RAJA typing)

If $\Gamma \vdash_{n'}^n e : C^r$ then there is an annotated version e° of the expression e with $|\Gamma|^\Psi \vdash_{u'}^u e^\circ \Leftarrow C^r$ for some $u \leq n$ and $u - u' \leq n - n'$ so that $\Gamma <: |\Gamma|^\Psi$.

- More details in the article "Efficient Type-Checking for Amortised Heap-Space Analysis" in CSL'09.

Correctness of the Algorithm

Lemma (Soundness of algorithmic RAJA typing)

If $\Delta^\Psi \vdash_{n'}^n e \Leftarrow C^\gamma$ then $\Delta^\Psi \vdash_{n'} |e| : C^\gamma$.

Lemma (Completeness of algorithmic RAJA typing)

If $\Gamma \vdash_{n'}^n e : C^r$ then there is an annotated version e° of the expression e with $|\Gamma|^\Psi \vdash_{u'}^u e^\circ \Leftarrow C^r$ for some $u \leq n$ and $u - u' \leq n - n'$ so that $\Gamma <: |\Gamma|^\Psi$.

- More details in the article "Efficient Type-Checking for Amortised Heap-Space Analysis" in CSL'09.

Correctness of the Algorithm

Lemma (Soundness of algorithmic RAJA typing)

If $\Delta^\Psi \frac{n}{n'} e \Leftarrow C^\gamma$ then $\Delta^\Psi \frac{n}{n'} |e| : C^\gamma$.

Lemma (Completeness of algorithmic RAJA typing)

If $\Gamma \frac{n}{n'} e : C^r$ then there is an annotated version e° of the expression e with $|\Gamma|^\Psi \frac{u}{u'} e^\circ \Leftarrow C^r$ for some $u \leq n$ and $u - u' \leq n - n'$ so that $\Gamma <: |\Gamma|^\Psi$.

- More details in the article "Efficient Type-Checking for Amortised Heap-Space Analysis" in CSL'09.

Subtyping algorithm

- With algorithmic views a sharing task can be reduced into a subtyping task: $\forall(C^r | C^{s_1}, C^{s_2}) \iff C^r <: C^{s_1+s_2}$.
- \Rightarrow It is enough to have a subtyping algorithm.
- $C^r <: D^s \iff (C^r, D^s)$ are elements of the greatest fixpoint of some operator.
- There is an efficient goal-directed algorithm for membership checking in greatest fixpoints, but only for a certain kind of operators (TPL, Pierce, 2002).
- The operator that describes subtyping is not of that kind.
- \Rightarrow We extended the given algorithm for all kinds of operators. (FICS'09)

Subtyping algorithm

- With algorithmic views a sharing task can be reduced into a subtyping task: $\forall(C^r | C^{s_1}, C^{s_2}) \iff C^r <: C^{s_1+s_2}$.
- \Rightarrow It is enough to have a subtyping algorithm.
- $C^r <: D^s \iff (C^r, D^s)$ are elements of the greatest fixpoint of some operator.
- There is an efficient goal-directed algorithm for membership checking in greatest fixpoints, but only for a certain kind of operators (TPL, Pierce, 2002).
- The operator that describes subtyping is not of that kind.
- \Rightarrow We extended the given algorithm for all kinds of operators. (FICS'09)

Subtyping algorithm

- With algorithmic views a sharing task can be reduced into a subtyping task: $\forall(C^r | C^{s_1}, C^{s_2}) \iff C^r <: C^{s_1+s_2}$.
- \Rightarrow It is enough to have a subtyping algorithm.
- $C^r <: D^s \iff (C^r, D^s)$ are elements of the greatest fixpoint of some operator.
- There is an efficient goal-directed algorithm for membership checking in greatest fixpoints, but only for a certain kind of operators (TPL, Pierce, 2002).
- The operator that describes subtyping is not of that kind.
- \Rightarrow We extended the given algorithm for all kinds of operators. (FICS'09)

Subtyping algorithm

- With algorithmic views a sharing task can be reduced into a subtyping task: $\forall(C^r | C^{s_1}, C^{s_2}) \iff C^r <: C^{s_1+s_2}$.
- \Rightarrow It is enough to have a subtyping algorithm.
- $C^r <: D^s \iff (C^r, D^s)$ are elements of the greatest fixpoint of some operator.
- There is an efficient goal-directed algorithm for membership checking in greatest fixpoints, but only for a certain kind of operators (TPL, Pierce, 2002).
- The operator that describes subtyping is not of that kind.
- \Rightarrow We extended the given algorithm for all kinds of operators. (FICS'09)

Subtyping algorithm

- With algorithmic views a sharing task can be reduced into a subtyping task: $\forall(C^r | C^{s_1}, C^{s_2}) \iff C^r <: C^{s_1+s_2}$.
- \Rightarrow It is enough to have a subtyping algorithm.
- $C^r <: D^s \iff (C^r, D^s)$ are elements of the greatest fixpoint of some operator.
- There is an efficient goal-directed algorithm for membership checking in greatest fixpoints, but only for a certain kind of operators (TPL, Pierce, 2002).
- The operator that describes subtyping is not of that kind.
- \Rightarrow We extended the given algorithm for all kinds of operators. (FICS'09)

Subtyping algorithm

- With algorithmic views a sharing task can be reduced into a subtyping task: $\forall(C^r | C^{s_1}, C^{s_2}) \iff C^r <: C^{s_1+s_2}$.
- \Rightarrow It is enough to have a subtyping algorithm.
- $C^r <: D^s \iff (C^r, D^s)$ are elements of the greatest fixpoint of some operator.
- There is an efficient goal-directed algorithm for membership checking in greatest fixpoints, but only for a certain kind of operators (TPL, Pierce, 2002).
- The operator that describes subtyping is not of that kind.
- \Rightarrow We extended the given algorithm for all kinds of operators. (FICS'09)

Implementation Features

- Typechecker for RAJA programs: prediction of free-list size.
- Interpreter with built-in calculation of free-list size.
- Datatypes
 - Integers
 - Booleans
 - Strings
- I/O features
 - Opening and closing files
 - Reading from files
 - Printing to the standard output
- Analysis gives an upper bound on the free-list size as a function of the size of the read files:

$$|\text{free-list}| \leq m + \sum_i (p_i * |\text{file}_i|)$$

- where p_i is the potential given to each line of the file file_i
- and m is a constant.

Implementation Features

- Typechecker for RAJA programs: prediction of free-list size.
- Interpreter with built-in calculation of free-list size.
- Datatypes
 - Integers
 - Booleans
 - Strings
- I/O features
 - Opening and closing files
 - Reading from files
 - Printing to the standard output
- Analysis gives an upper bound on the free-list size as a function of the size of the read files:

$$|\text{free-list}| \leq m + \sum_i (p_i * |\text{file}_i|)$$

- where p_i is the potential given to each line of the file file_i
- and m is a constant.

Implementation Features

- Typechecker for RAJA programs: prediction of free-list size.
- Interpreter with built-in calculation of free-list size.
- Datatypes
 - Integers
 - Booleans
 - Strings
- I/O features
 - Opening and closing files
 - Reading from files
 - Printing to the standard output
- Analysis gives an upper bound on the free-list size as a function of the size of the read files:

$$|\text{free-list}| \leq m + \sum_i (p_i * |\text{file}_i|)$$

- where p_i is the potential given to each line of the file file_i
- and m is a constant.

Implementation Features

- Typechecker for RAJA programs: prediction of free-list size.
- Interpreter with built-in calculation of free-list size.
- Datatypes
 - Integers
 - Booleans
 - Strings
- I/O features
 - Opening and closing files
 - Reading from files
 - Printing to the standard output
- Analysis gives an upper bound on the free-list size as a function of the size of the read files:

$$|\text{free-list}| \leq m + \sum_i (p_i * |\text{file}_i|)$$

- where p_i is the potential given to each line of the file file_i
- and m is a constant.

Conclusions and further work

- Conclusions
 - Our type-based analysis encompasses:
 - ① Objects
 - ② Inheritance
 - ③ Downcast
 - ④ Imperative update
 - ⑤ Aliasing
 - ⑥ Circular data
 - The (improved) implementation allows:
 - Testing the system with bigger programs and bigger inputs
 - Improving the system
- Further work
 - Elimination of the annotations of the variable occurrences.
 - Type inference
 - More examples: Iterators on lists, etc.
 - Making the system more expressive (maybe with type states).

Conclusions and further work

- Conclusions

- Our type-based analysis encompasses:

- ① Objects
- ② Inheritance
- ③ Downcast
- ④ Imperative update
- ⑤ Aliasing
- ⑥ Circular data

- The (improved) implementation allows:

- Testing the system with bigger programs and bigger inputs
- Improving the system

- Further work

- Elimination of the annotations of the variable occurrences.
- Type inference
- More examples: Iterators on lists, etc.
- Making the system more expressive (maybe with type states).