Gradual Ownership Types

Ilya Sergey Dave Clarke

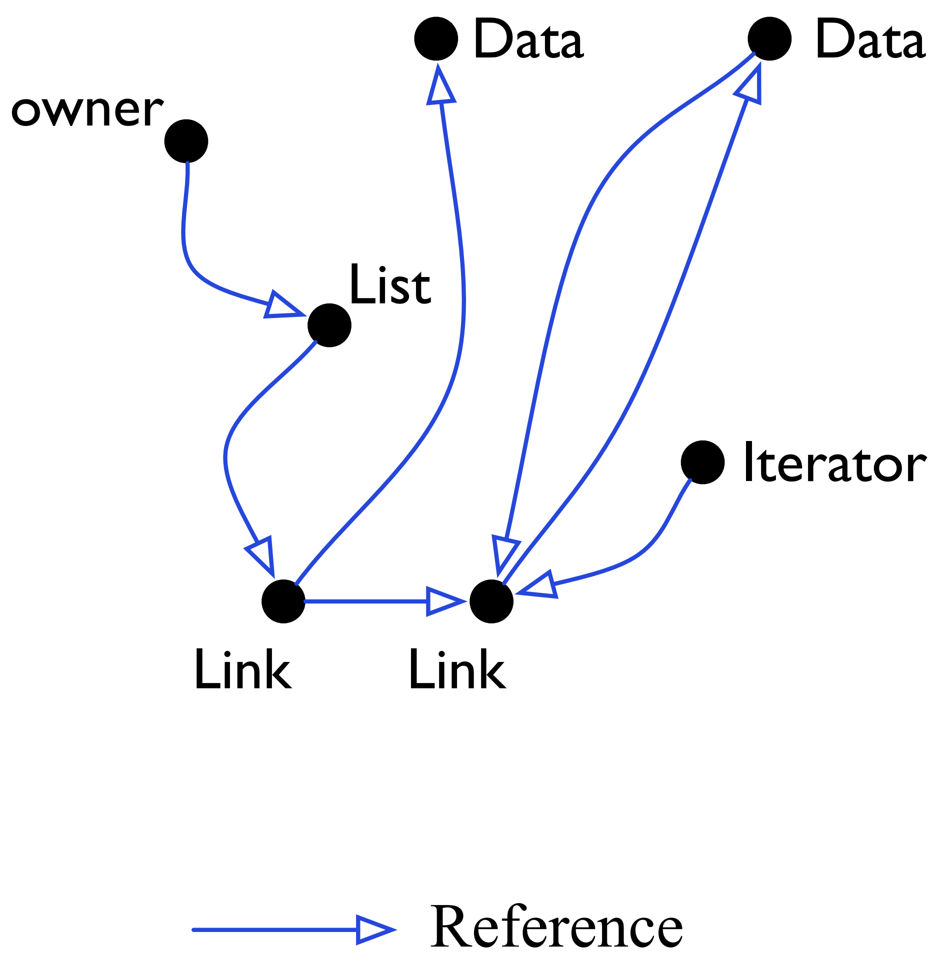
KATHOLIEKE UNIVERSITEIT
LEUVEN

ESOP 2012

Ownership Types

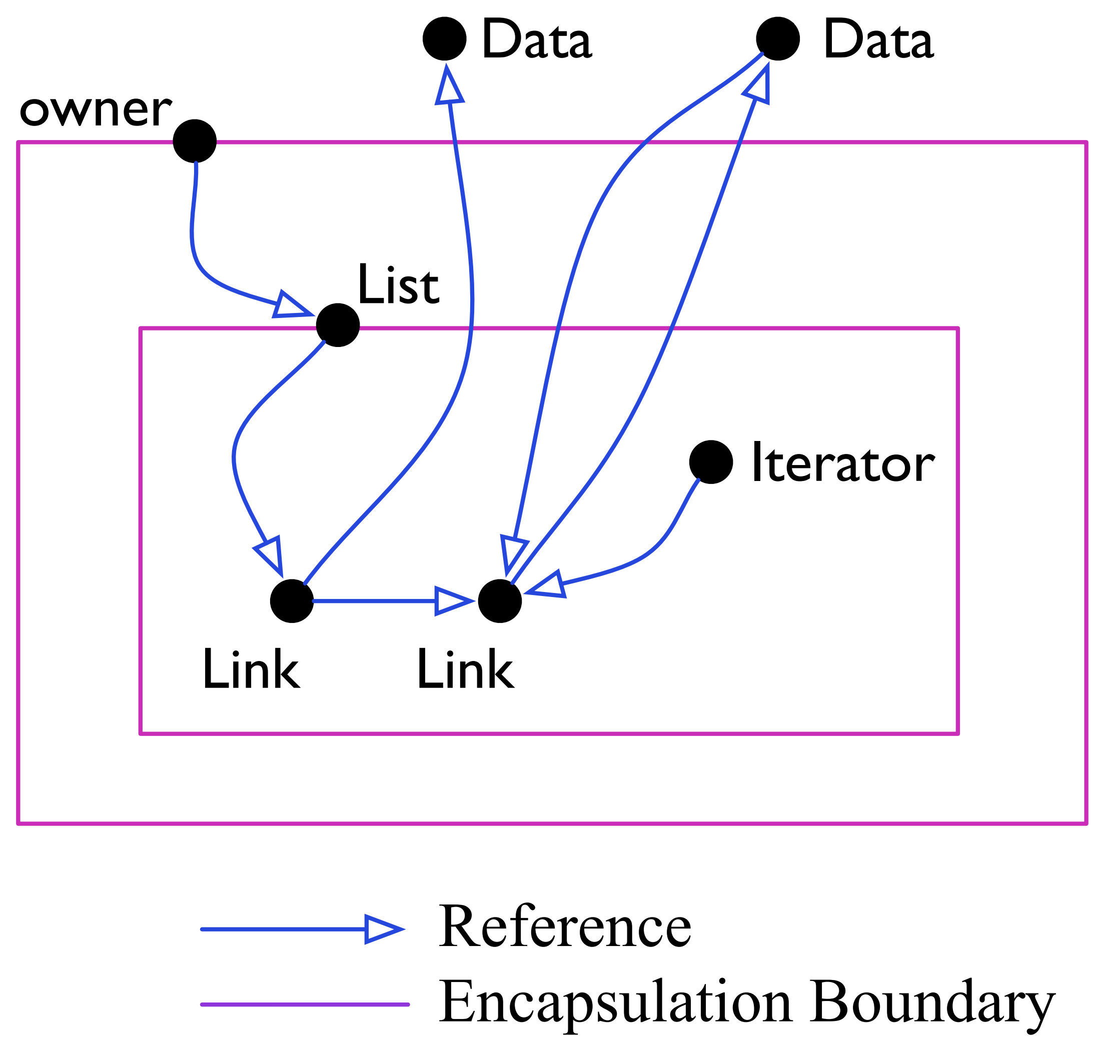
(a gradual introduction)

```
class List {
  Link head;
  void add(Data d) {
    head = new Link(head, d);
  }
  Iterator makeIterator() {
    return new Iterator(head);
  }
}
class Link {
  Link next;
  Data data;
  Link(Link next, Data data) {
    this.next = next; this.data = data;
  }
}
class Iterator {
  Link current;
  Iterator(Link first) {
    current = first;
  }
  void next() { current = current.next; }
  Data elem() { return current.data; }
  boolean done() {
    return (current == null);
  }
}
```



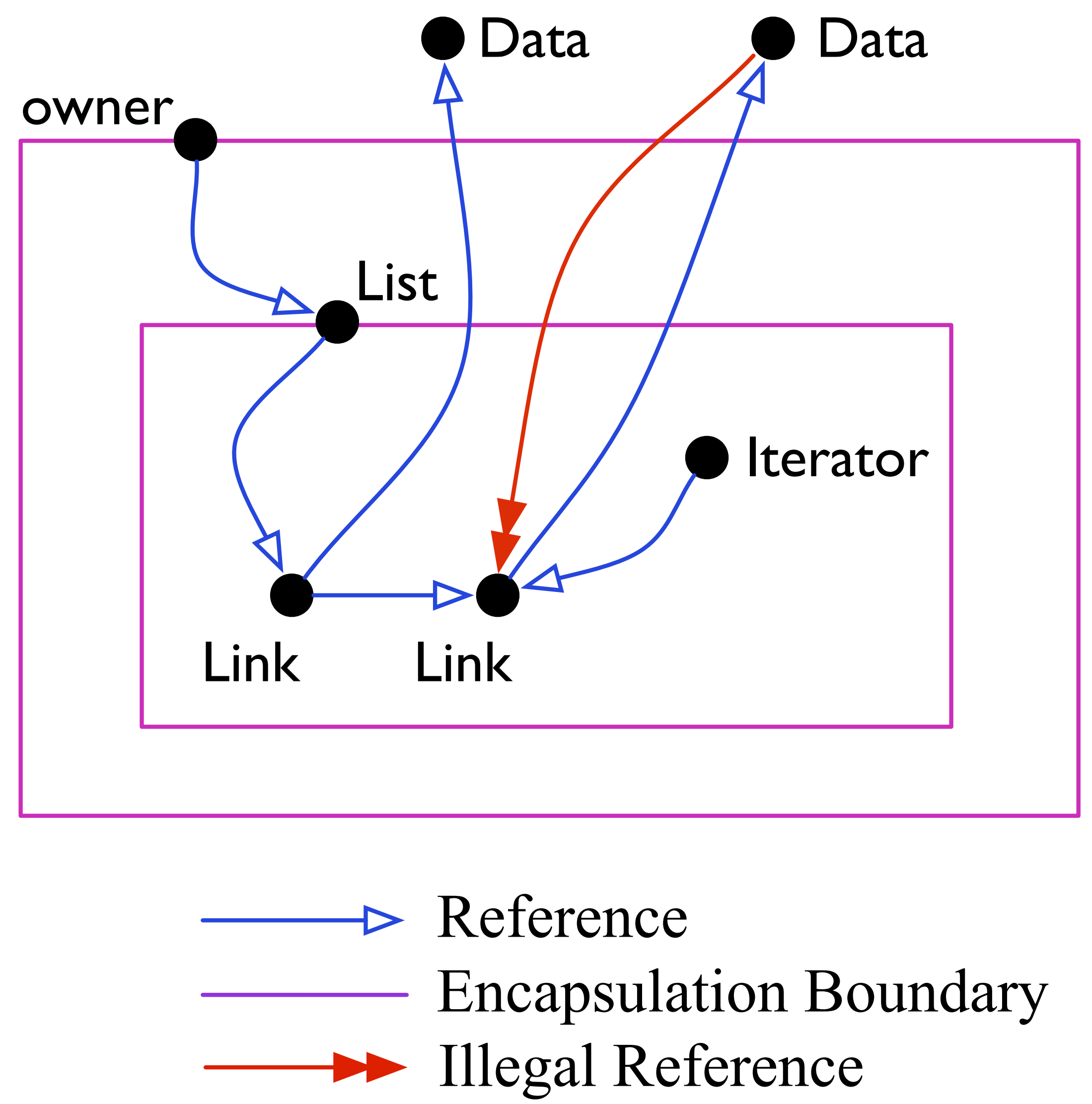
Ownership Types

```
class List {
  Link head;
  void add(Data d) {
    head = new Link(head, d);
  }
  Iterator makeIterator() {
    return new Iterator(head);
  }
}
class Link {
  Link next;
  Data data;
  Link(Link next, Data data) {
    this.next = next; this.data = data;
  }
}
class Iterator {
  Link current;
  Iterator(Link first) {
    current = first;
  }
  void next() { current = current.next; }
  Data elem() { return current.data; }
  boolean done() {
    return (current == null);
  }
}
```



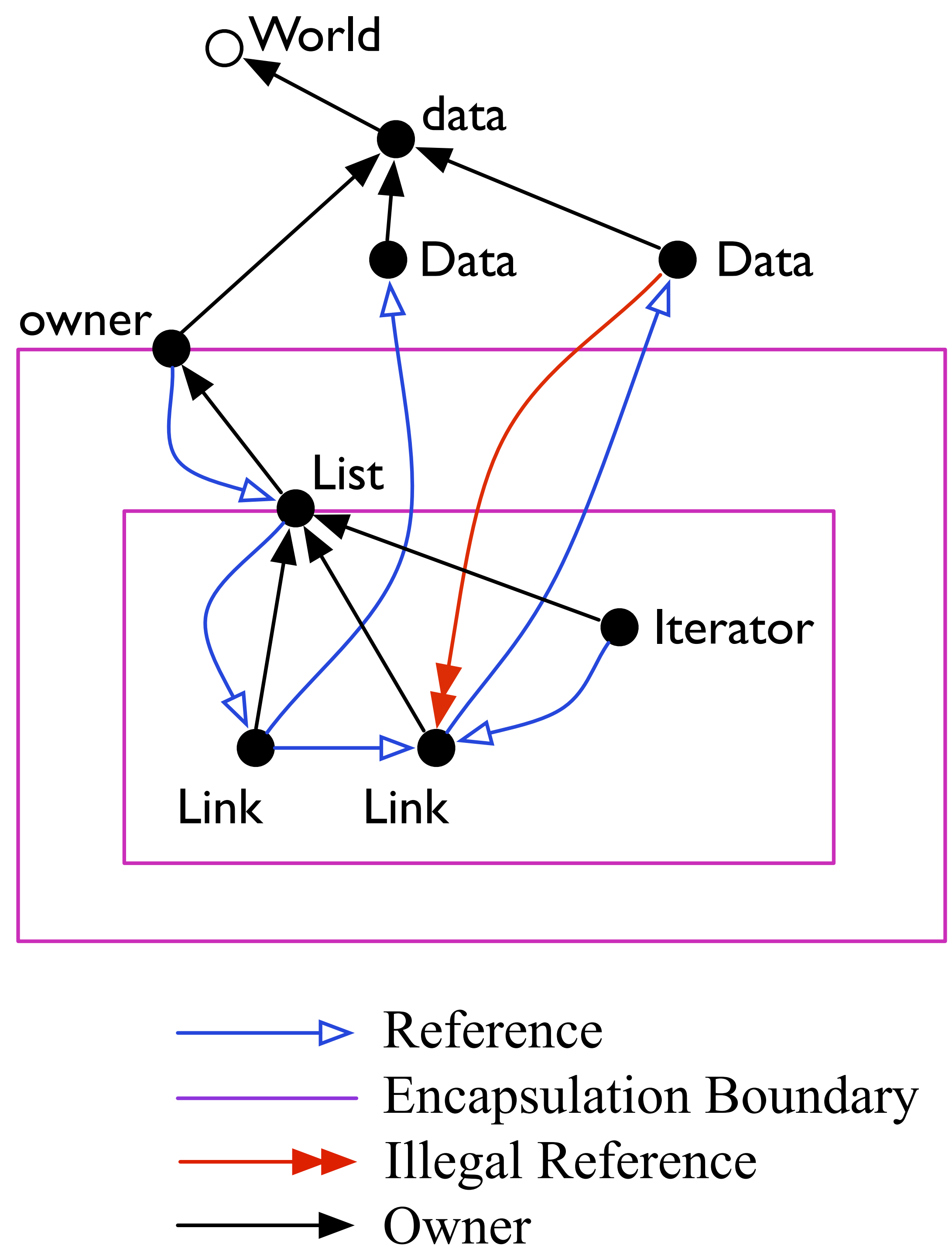
Ownership Types

```
class List {
  Link head;
  void add(Data d) {
    head = new Link(head, d);
  }
  Iterator makeIterator() {
    return new Iterator(head);
  }
}
class Link {
  Link next;
  Data data;
  Link(Link next, Data data) {
    this.next = next; this.data = data;
  }
}
class Iterator {
  Link current;
  Iterator(Link first) {
    current = first;
  }
  void next() { current = current.next; }
  Data elem() { return current.data; }
  boolean done() {
    return (current == null);
  }
}
```



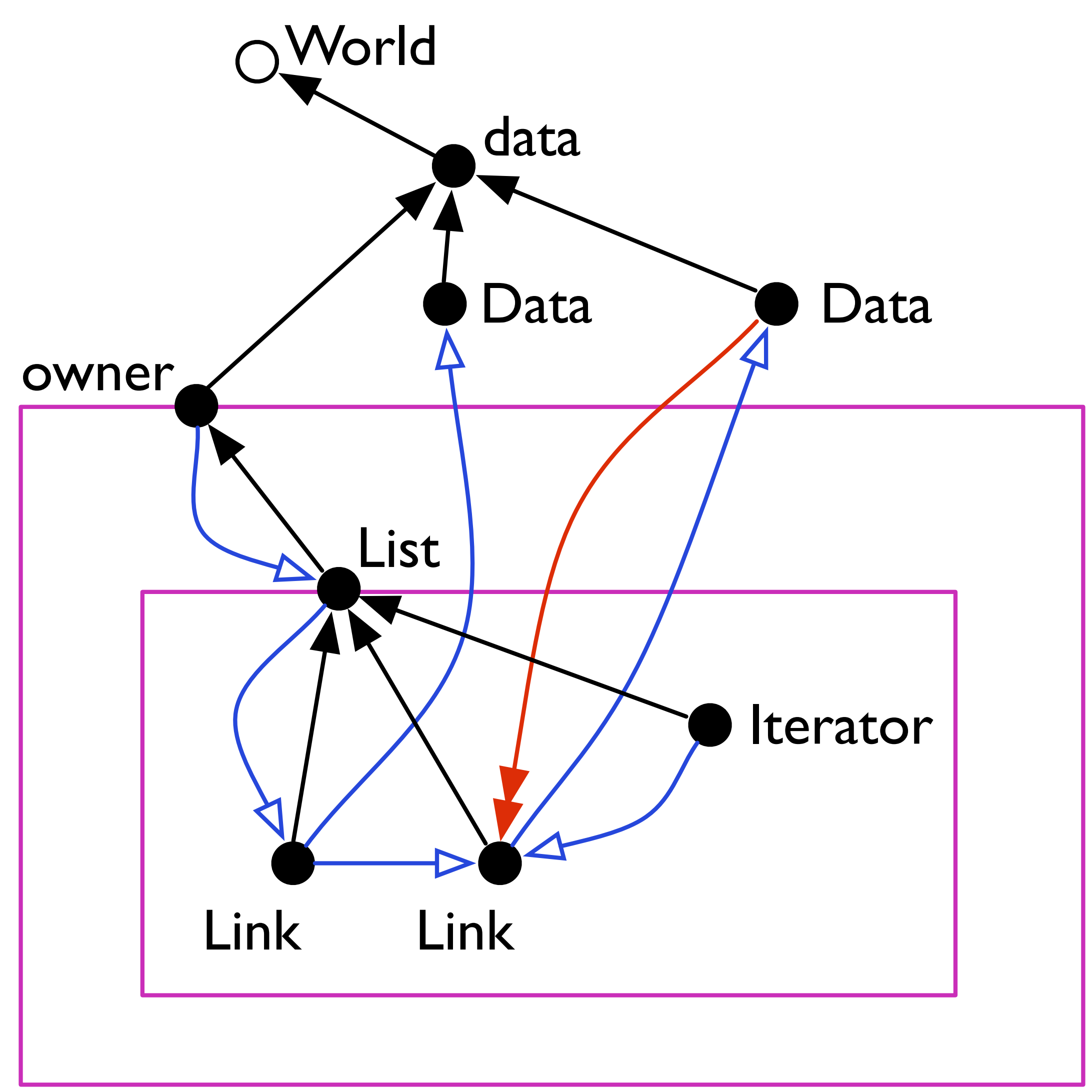
Ownership Types

```
class List {
  Link head;
  void add(Data d) {
    head = new Link(head, d);
  }
  Iterator makeIterator() {
    return new Iterator(head);
  }
}
class Link {
  Link next;
  Data data;
  Link(Link next, Data data) {
    this.next = next; this.data = data;
  }
}
class Iterator {
  Link current;
  Iterator(Link first) {
    current = first;
  }
  void next() { current = current.next; }
  Data elem() { return current.data; }
  boolean done() {
    return (current == null);
  }
}
```



Ownership Types

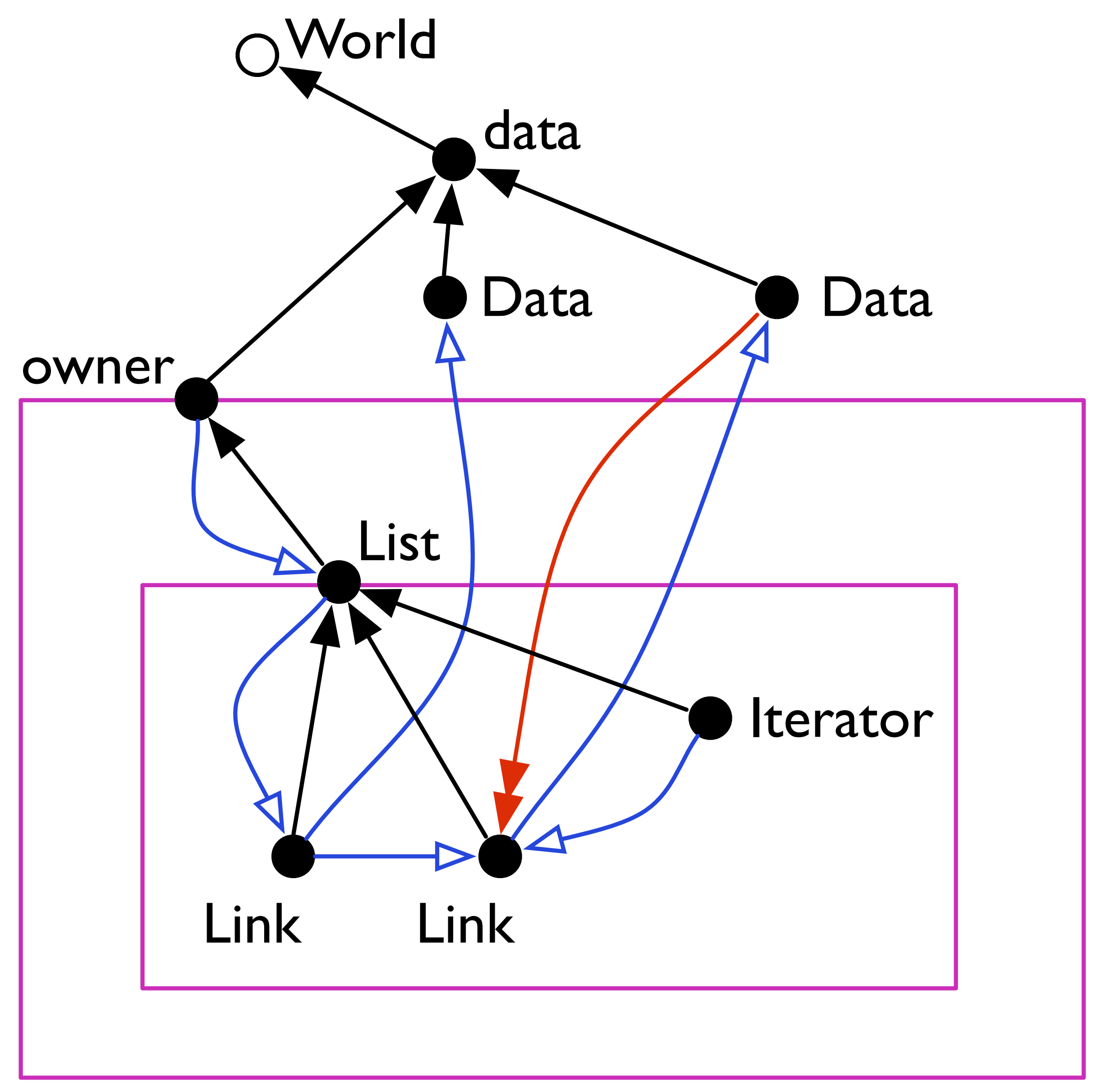
```
class List {
  Link head;
  void add(Data d) {
    head = new Link(head, d);
  }
  Iterator makeIterator() {
    return new Iterator(head);
  }
}
class Link {
  Link next;
  Data data;
  Link(Link next, Data data) {
    this.next = next; this.data = data;
  }
}
class Iterator {
  Link current;
  Iterator(Link first) {
    current = first;
  }
  void next() { current = current.next; }
  Data elem() { return current.data; }
  boolean done() {
    return (current == null);
  }
}
```



Owners-as-Dominators
(OAD)

Ownership Types

```
class List<owner, data> {
  Link head<this, data>;
  void add(Data<data> d) {
    head = new Link<this, data>(head, d);
  }
  Iterator<this, data> makeIterator() {
    return new Iterator<this, data>(head);
  }
}
class Link<owner, data> {
  Link<owner, data> next;
  Data<data> data;
  Link(Link<owner, data> next, Data<data> data) {
    this.next = next; this.data = data;
  }
}
class Iterator<owner, data> {
  Link<owner, data> current;
  Iterator(Link<owner, data> first) {
    current = first;
  }
  void next() { current = current.next; }
  Data<data> elem() { return current.data; }
  boolean done() {
    return (current == null);
  }
}
```



Owners-as-Dominators (OAD)

Good things about
Ownership Types

- data-race freedom [Boyapati-Rinard:OOPSLA01]
- disjointness of effects [Clarke-Drossopoulou:OOPSLA02]
- various confinement properties [Vitek-Bokowski:OOPSLA99]
- effective memory management [Boyapati-et-al:PLDI03]
- modular reasoning about aliasing [Müller:VSTTE05]

Bad things about
Ownership Types

Verbose
and
Restrictive

Bad things about

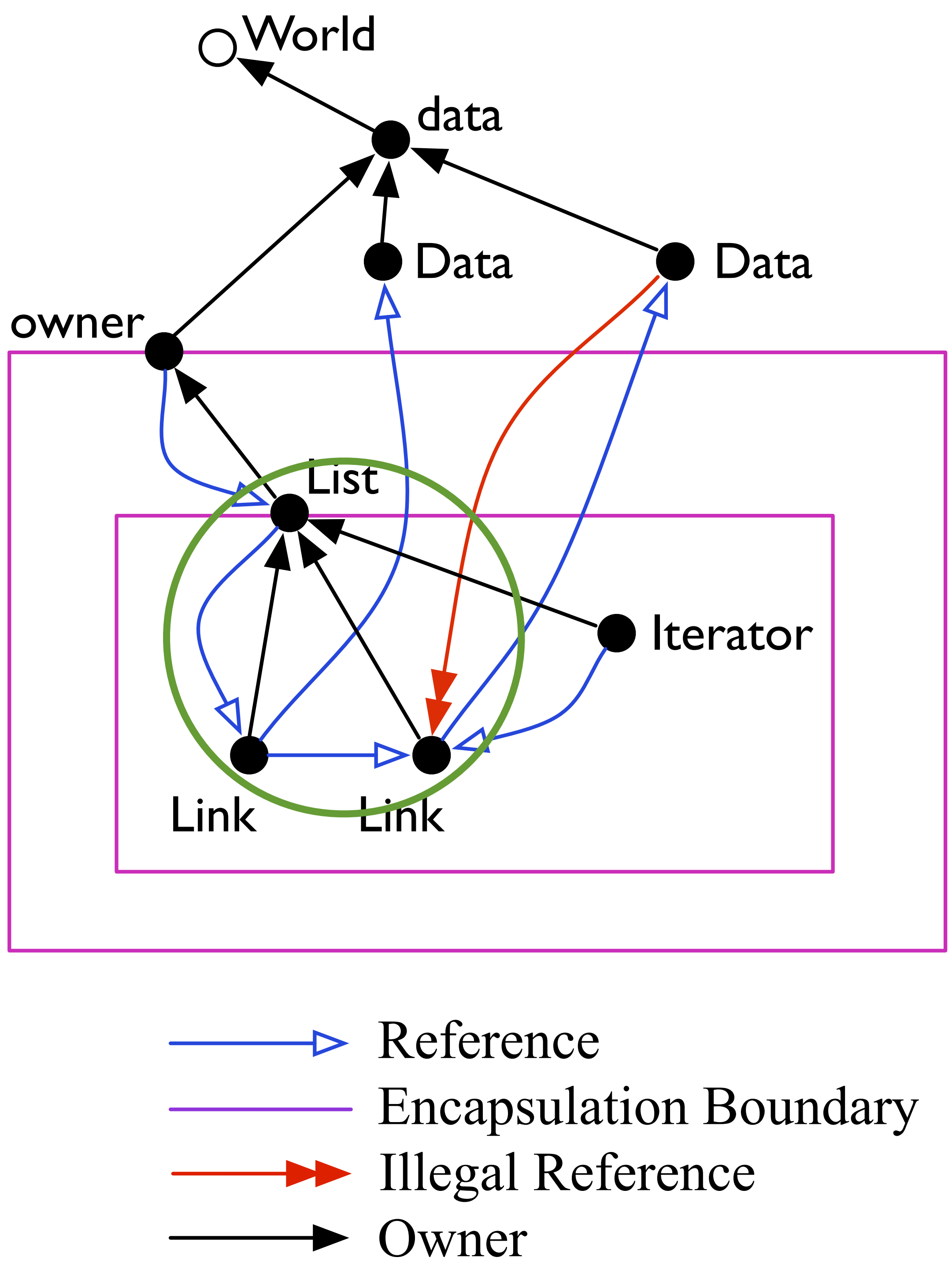
Ownership Types

```
class List {
  Link head;
  void add(Data d) {
    head = new Link(head, d);
  }
  Iterator makeIterator() {
    return new Iterator(head);
  }
}
class Link {
  Link next;
  Data data;
  Link(Link next, Data data) {
    this.next = next; this.data = data;
  }
}
class Iterator {
  Link current;
  Iterator(Link first) {
    current = first;
  }
  void next() { current = current.next; }
  Data elem() { return current.data; }
  boolean done() {
    return (current == null);
  }
}
```

```
class List<owner, data> {
  Link head<this, data>;
  void add(Data<data> d) {
    head = new Link<this, data>(head, d);
  }
  Iterator<this, data> makeIterator() {
    return new Iterator<this, data>(head);
  }
}
class Link<owner, data> {
  Link<owner, data> next;
  Data<data> data;
  Link(Link<owner, data> next, Data<data> data) {
    this.next = next; this.data = data;
  }
}
class Iterator<owner, data> {
  Link<owner, data> current;
  Iterator(Link<owner, data> first) {
    current = first;
  }
  void next() { current = current.next; }
  Data<data> elem() { return current.data; }
  boolean done() {
    return (current == null);
  }
}
```

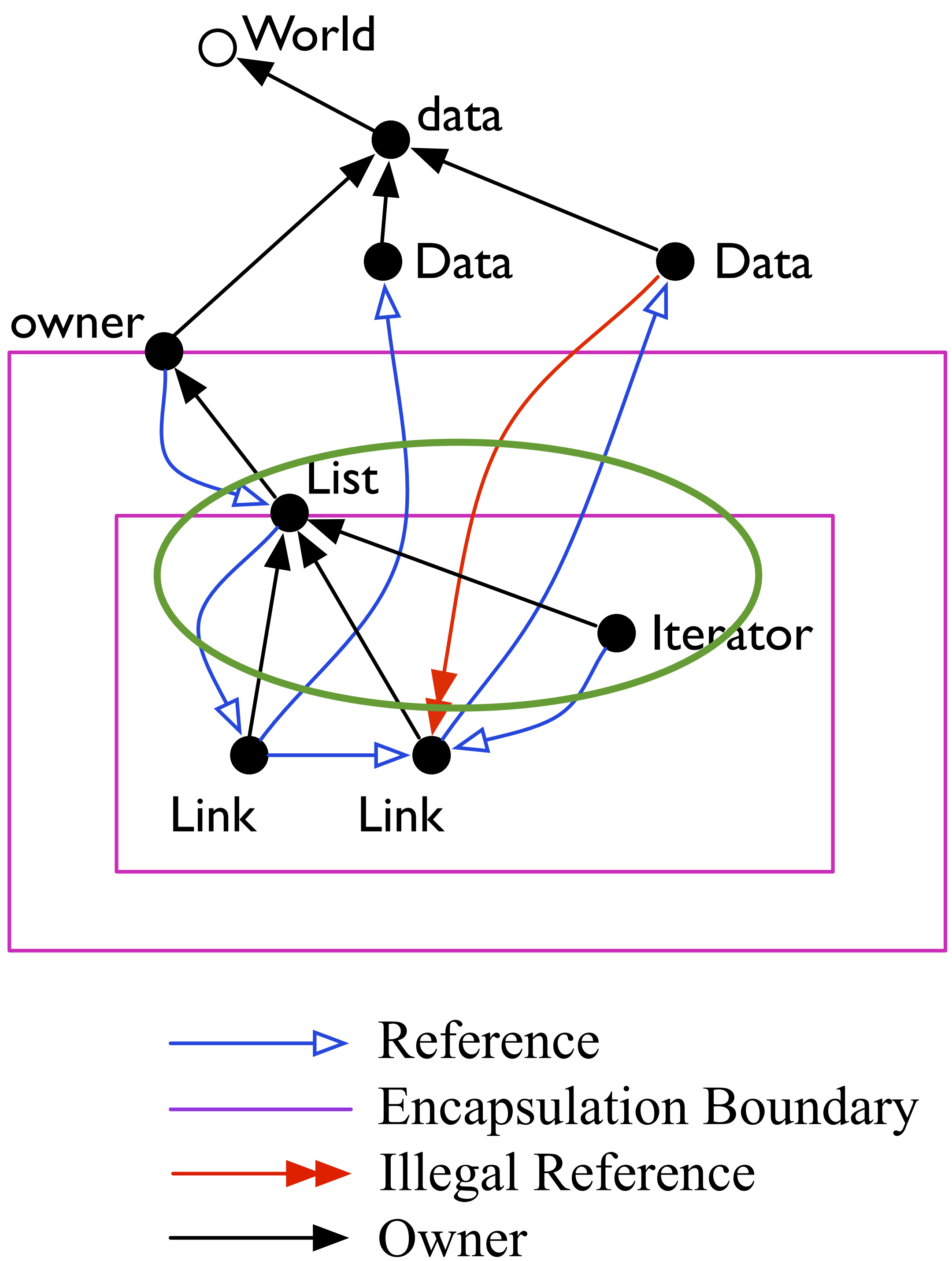
15 annotations

The intention




```
class List<owner, data> {
  Link head<this, data>;
  void add(Data<data> d) {
    head = new Link<this, data>(head, d);
  }
  Iterator<this, data> makeIterator() {
    return new Iterator<this, data>(head);
  }
}
class Link<owner, data> {
  Link<owner, data> next;
  Data<data> data;
  Link(Link<owner, data> next, Data<data> data) {
    this.next = next; this.data = data;
  }
}
class Iterator<owner, data> {
  Link<owner, data> current;
  Iterator(Link<owner, data> first) {
    current = first;
  }
  void next() { current = current.next; }
  Data<data> elem() { return current.data; }
  boolean done() {
    return (current == null);
  }
}
```

The intention




```
class List<owner, data> {
  Link head<this, data>;
  void add(Data<data> d) {
    head = new Link<this, data>(head, d);
  }
  Iterator<this, data> makeIterator() {
    return new Iterator<this, data>(head);
  }
}
class Link<owner, data> {
  Link<owner, data> next;
  Data<data> data;
  Link(Link<owner, data> next, Data<data> data) {
    this.next = next; this.data = data;
  }
}
class Iterator<owner, data> {
  Link<owner, data> current;
  Iterator(Link<owner, data> first) {
    current = first;
  }
  void next() { current = current.next; }
  Data<data> elem() { return current.data; }
  boolean done() {
    return (current == null);
  }
}
```

Can we implement
the same intention with a
fewer amount of annotations?

A few analogies

- properties of data ~ types
- OAD invariant ~ *more precise* types

Untyped program → Typed program → Even more typed program

From untyped to typed - I

Type Inference

- SmallTalk [Palsberg-Schwartzbach:OOPSLA91]
- Ruby [Fur-An-Foster-Hicks:SAC09, An-Chaudhuri-Foster-Hicks:POPL11]
- JavaScript [Jensen-Møller-Thiemann:SAS10, Guha-al:ESOP11]

Ownership (Type) Inference

- Profiling-based approaches
  - Wren:MS03, Dietl-Müller:IWACO'07...
- Static CFA-based approaches
  - **Ownership Types:** Moelius-Souter:MASPLAS04, Huang-Milanova:IWACO11, Milanova-Vitek:TOOLS10, Milanova-Liu:TR10, Dietl-Ernst-Muller:ECOOP11 ...
  - **Ownership properties:** Geilman-Poetzsch-Heffer:IWACO11, Ma-Foster:OOPSLA07, Greenfieldboyce:Foster:OOPSLA07, Aldrich-Kostadinov-Chambers:OOPSLA02 ...

Why not ownership inference?

- Correctness of inference with respect to the type system is *hard* to prove
- Inferred results might be *imprecise* and *difficult* to analyze

From untyped to typed - II

(Partially) relying on dynamic checks

- Gradual Typing [Siek-Taha:ECOOP07, Herman-Tomb-Flanagan:TFP07]
- Hybrid Types [Flanagan:POPL06]
- Contracts [Findler-Felleisen:ICFP02, Gray-Findler-Flatt:OOPSLA05]
- Like types [Wrigstad-ZappaNardelli-Lebresne-Östlund-Vitek:POPL10]

*

- Dynamic ownership [Gordon-Noble:DLS07]
  - No relation to the type system

* Detailed comparison: Greenberg-Pierce-Weirich:POPL10

Gradual Types

- Programmers may omit type annotations and run the program immediately
  - Run-time checks are *inserted* to ensure *type safety*
- Programmers may add type annotations to increase static checking
  - When all sites are annotated, *all* type errors are caught at compile-time

Gradual Ownership Types

A syntactic type parametrized with owners:

`C<owner, outer>`

Some owners *might* be *unknown:*

`C<?, outer>`

Or even all of them:

`C ≡ C<?, ?>`

Type equality: types T_1 and T_2 are *equal*:

```
C<owner, outer> = C<owner, outer>
```

Type equality: types T_1 and T_2 are *consistent*

```
C<owner, ?> ~ C<?, outer>
```

I.e., T_1 and T_2 *might* correspond to the *same* runtime values

Traditional Subtyping

```
class D<MyOwner> {...}

class C<Owner1, Owner2> extends D<Owner1> {...}
```

Subtyping: T_1 is a *subtype* of T_2

```
C<owner, outer> ≤ D<owner>
```

$\boxed{E;B \vdash t \leq t'}$

$$\frac{E;B \vdash t}{E;B \vdash t \leq t} \ \text{(SUB-REFL)}$$

Reflexive

$$\frac{E;B \vdash t \leq t' \quad E;B \vdash t' \leq t''}{E;B \vdash t \leq t''} \ \text{(SUB-TRANS)}$$

Transitive

$$\frac{E;B \vdash c\langle\sigma\rangle \quad \texttt{class}\ c\langle\alpha_{i\in 1..n}\rangle\ \texttt{extends}\ c'\langle r_{i\in 1..n'}\rangle\{\ldots\}}{E;B \vdash c\langle\sigma\rangle \leq c'\langle\sigma(r_i)_{i\in 1..n'}\rangle} \ \text{(SUB-CLASS)}$$

Nominal

Gradual Subtyping

```
class D<MyOwner> {...}

class C<Owner1, Owner2> extends D<Owner1> {...}
```

$$
\begin{tikzcd}
\texttt{D<?>} \arrow[r, "\sim"] & \texttt{D<owner>} \\
\texttt{C<?, outer>} \arrow[u, "\leq"] \arrow[ur, "\lesssim"'] &
\end{tikzcd}
$$

$$\texttt{C<?, outer>} \lesssim \texttt{D<owner>}$$

Static semantics

$\boxed{E;B \vdash p \sim p'}$

Consistent owners

$$\frac{E;B \vdash p}{E;B \vdash p \sim p}\ (\textsc{con-refl}) \qquad \frac{E;B \vdash p}{E;B \vdash\ ? \sim p}\ (\textsc{con-right}) \qquad \frac{E;B \vdash p}{E;B \vdash p \sim ?}\ (\textsc{con-left})$$

$$\frac{E;B \vdash p \quad E;B \vdash x^{c.i}}{E;B \vdash p \sim x^{c.i}}\ (\textsc{con-dependent1}) \qquad \frac{E;B \vdash p \quad E;B \vdash x^{c.i}}{E;B \vdash x^{c.i} \sim p}\ (\textsc{con-dependent2})$$

$\boxed{E;B \vdash t \leq t'}$

Traditional subtyping

$$\frac{E;B \vdash t}{E;B \vdash t \leq t}\ (\textsc{sub-refl}) \qquad \frac{E;B \vdash t \leq t' \quad E;B \vdash t' \leq t''}{E;B \vdash t \leq t''}\ (\textsc{sub-trans})$$

$$\frac{E;B \vdash c\langle\sigma\rangle \qquad \texttt{class}\ c\langle\alpha_{i\in 1..n}\rangle\ \texttt{extends}\ c'\langle r_{i\in 1..n'}\rangle\{\ldots\}}{E;B \vdash c\langle\sigma\rangle \leq c'\langle\sigma(r_i)_{i\in 1..n'}\rangle}\ (\textsc{sub-class})$$

$\boxed{E;B \vdash t \sim t'}$ $\qquad$ $\boxed{E;B \vdash t \lesssim t'}$ $\qquad$ $\boxed{E;B \vdash t}$ "Good type"

$$\frac{E;B \vdash c\langle p_{i\in 1..n}\rangle \quad E;B \vdash c\langle q_{i\in 1..n}\rangle \quad p_i \sim q_i \forall i \in 1..n}{E;B \vdash c\langle p_{i\in 1..n}\rangle \sim c\langle q_{i\in 1..n}\rangle}\ (\textsc{con-type})$$

$$\frac{E;B \vdash c\langle\sigma\rangle \leq c'\langle\sigma'\rangle \quad E;B \vdash c'\langle\sigma'\rangle \sim c'\langle\sigma''\rangle}{E;B \vdash c\langle\sigma\rangle \lesssim c'\langle\sigma''\rangle}\ (\textsc{grad-sub})$$

$$\frac{\mathsf{arity}(c) = n \quad E;B \vdash p_1 \preceq p_i \quad \forall i \in 1..n}{E;B \vdash c\langle p_{i\in 1..n}\rangle}\ (\textsc{g-type})$$

Consistent types $\qquad$ "Gradual Subtyping"

Example I

```
List list;  // = List<?,?>
list = new List<p, world>();
list = new List<this, world>();
List<p, world> newList = list;
```

Dangerous assignment!

Example I

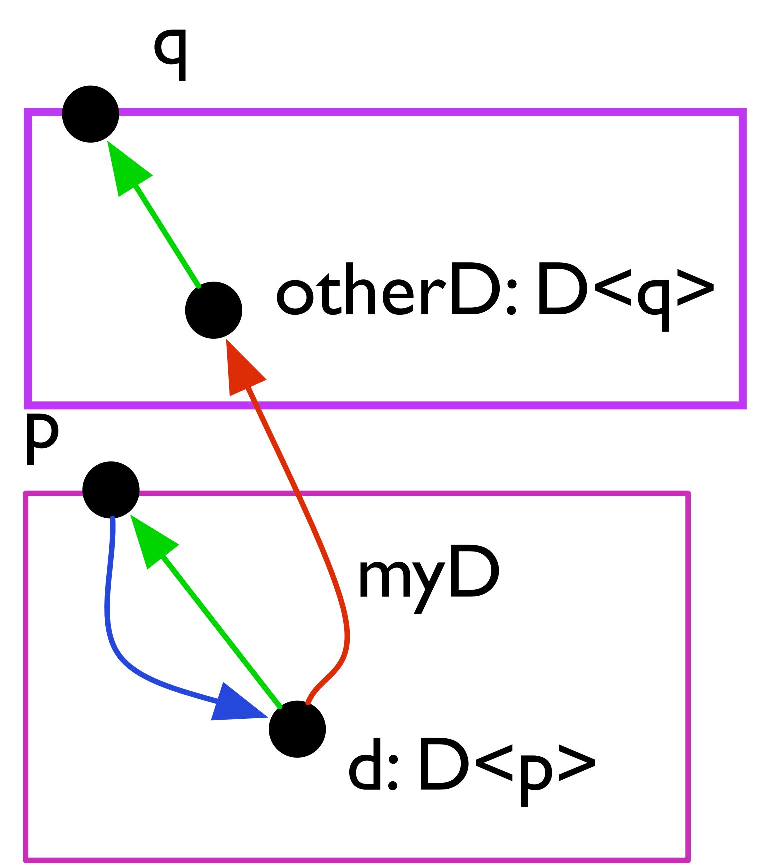
```
List list;  // = List<?,?>
list = new List<p, world>();
list = new List<this, world>();
List<p, world> newList =
                   (List<p, world>)list;
```

Dynamic type cast inserted

Example II

```
class D<owner> {
  D myD; // = D<?>
}
...
D<q> otherD = ...;
D<p> d = new D<p>();
d.myD = otherD;
```

Possible OAD violation

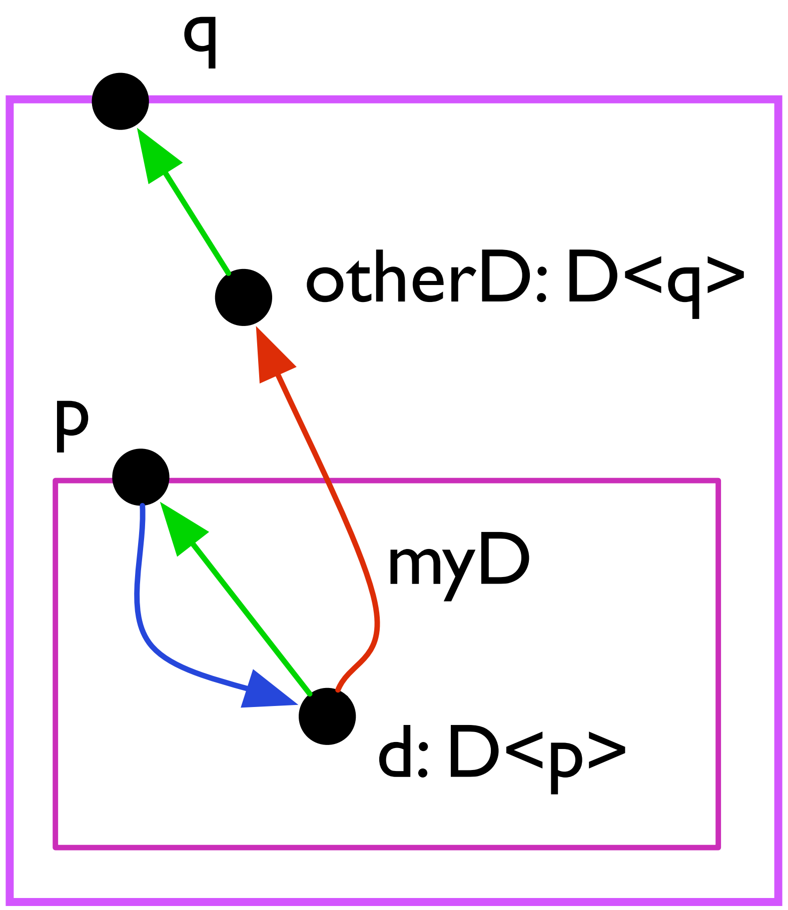


Example II

```
class D<owner> {
  D myD; // = D<?>
}
...
D<q> otherD = ...;
D<p> d = new D<p>();
d.myD =
      bcheck(d, otherD);
```

Boundary check inserted

$$\mathtt{d} \prec^{*} \underline{\mathit{owner}}(\mathtt{otherD})$$



Type-directed compilation

Dynamic casts and *boundary checks* are inserted basing on type information.

$\boxed{E;B \vdash b : s}$

$$\text{(T-NEW)}\quad \frac{E;B \vdash c\langle r_{i\in 1..n}\rangle}{E;B \vdash \texttt{new}\, c\langle r_{i\in 1..n}\rangle : c\langle r_{i\in 1..n}\rangle}$$

$$\text{(T-LKP)}\quad \frac{E;B \vdash z : c\langle \sigma\rangle \quad \mathcal{F}_c(f) = t}{E;B \vdash z.f : \sigma_z(t)}$$

$$\text{(T-LET)}\quad \frac{E;B \vdash b : t \quad E, x : \mathsf{fill}(x,t);B \vdash e : s}{E;B \vdash \texttt{let}\, x = b\, \texttt{in}\, e : s}$$

Field update

Method call

$$\text{(T-UPD)}\quad \frac{E;B \vdash z : c\langle \sigma\rangle \quad \mathcal{F}_c(f) = t \quad E;B \vdash y : s \quad E;B \vdash s \lesssim \sigma_z(t)}{E;B \vdash z.f = y : \sigma_z(t)}$$

$$\text{(T-CALL)}\quad \frac{E;B \vdash y : s \quad \mathcal{MT}_c(m) = (y', t \to t') \quad E;B \vdash z : c\langle \sigma\rangle \quad E;B \vdash s \lesssim \sigma_z(t) \quad \sigma' \equiv \sigma \uplus \{y' \mapsto y\}}{E;B \vdash z.m(y) : \sigma'_z(t')}$$

$$\text{(VAL-}w\text{)}\quad \frac{E;B \vdash \diamond\, w : s \in E}{E;B \vdash w : s}$$

$$\text{(VAL-NULL)}\quad \frac{E;B \vdash t}{E;B \vdash \texttt{null} : t}$$

$\boxed{E \vdash t'\, m(t\ y)\ \{e\}}$

$\boxed{\vdash P; e}$

Method return

$$\text{(METHOD)}\quad \frac{E, y : \mathsf{fill}(y,t) \vdash e : s \quad E \vdash s \lesssim t'}{E \vdash t'\, m(t\ y)\ \{e\}}$$

$$\text{(PROGRAM)}\quad \frac{\vdash class_j \ \ \forall class_j \in P \quad E \vdash e : t}{E \vdash P; e}$$

Gradual subtyping might cause check insertion

Type-directed compilation

Two-staged program translation

- Insert *dynamic casts* to <u>*coerce*</u> types
  - Type consistency $\Rightarrow$ Type equality
- Insert *boundary checks* when the invariant can be violated
  - Check $\prec$ for unknown owners

Minimal amount of annotations

```
class List<owner, data> {
  Link head<this, data>;
  void add(Data<data> d) {
    head = new Link<this, data>(head, d);
  }
  Iterator<this, data> makeIterator() {
    return new Iterator<this, data>(head);
  }
}
class Link<owner, data> {
  Link<owner, data> next;
  Data<data> data;
  Link(Link<owner, data> next, Data<data> data) {
    this.next = next; this.data = data;
  }
}
class Iterator<owner, data> {
  Link<owner, data> current;
  Iterator(Link<owner, data> first) {
    current = first;
  }
  void next() { current = current.next; }
  Data<data> elem() { return current.data; }
  boolean done() {
    return (current == null);
  }
}
```

5 annotations are needed to indicate the intention:

3 annotations to indicate the parametrization

2 annotations for instance owners

10 annotations are optional

Gradual Typing and Compilation (informally)*

Theorem 1:
No unknown owners ⇒ no dynamic casts

Corollary :
No unknown owners ⇒ static invariant guaranty

(And also, no runtime overhead and failed casts)

Theorem 2:
A (gradually) well-typed program is compiled into a (statically) well-typed program.

* Formal treatment + proofs at `http://people.cs.kuleuven.be/~ilya.sergey/gradual`

Type safety result (informally)

Theorem 3:
A (statically) well-typed program does not violate the OAD invariant but might fail on a dynamic check.

Corollary:
A gradually well-typed program, being compiled, does not violate the OAD invariant.

"Well-typed programs don't go wrong"
Milner, 1978

Pitfalls of the approach

- Static safety is traded for dynamic checks
- Memory overhead
  - References to owners are stored in objects
- Runtime overhead
  - dynamic boundary checks and type casts

Implementation*

- Implemented in JastAddJ [Ekman-Hedin:OOPSLA07]
  - Extended JastAddJ compiler for Java 1.4
- 2,600 LOC (not including tests and comments)
- Check insertion ⇒ compilation warning
- Source-to-source translation

* Available from http://github.com/ilyasergey/Gradual-Ownership

Experience

- Java Collection Framework (JDK 1.4.2)
  - 46 source files, ~8,200 LOC
- Securing inner `Entries` of collections
- Questions addressed:
  - How many annotations are needed minimally?
  - What is the execution cost?
  - How many annotations for full static checking?

Experience

- Minimal amount of annotations
  - `LinkedList` - 17
  - `LinkedMap` - 15
- Performance overhead
  - ~1.5-2 times (for extensive updates)
- Full migration
  - `LinkedList` - yes, 34 annotations
  - `LinkedMap` - no, because of static
    - (best - 28 annotations)

Summary

Gradual Ownership Types

- An alternative to ownership inference
- Combines static and dynamic ownership checks, but allows *full static* safety
- Type-directed compilation
- Minimal annotations are unavoidable
- A tradeoff between verbosity and safety

Thanks