Tunable Static Inference for Generic Universe Types

Werner Dietl

Michael Ernst & Peter Müller

UNIVERSITY of WASHINGTON
COMPUTER SCIENCE & ENGINEERING

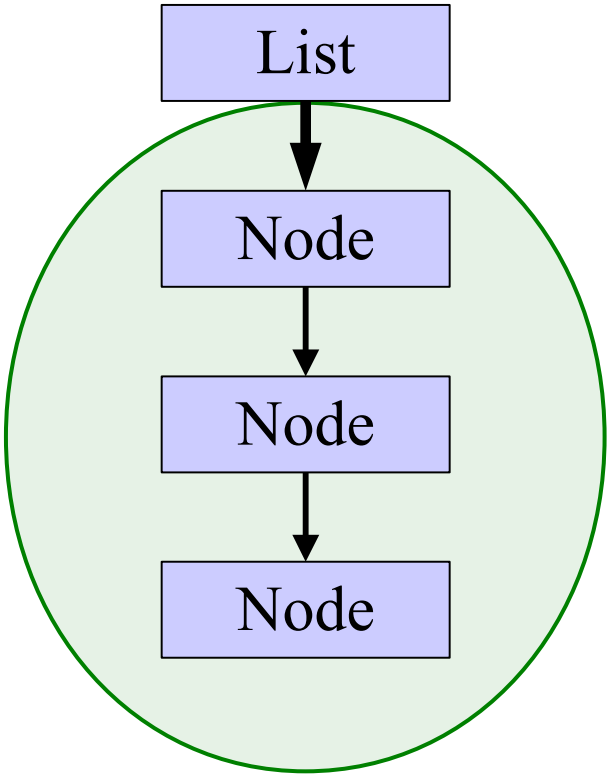
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Generic Universe Types (GUT)

- Lightweight ownership type system
- Heap topology
- Owner-as-Modifier encapsulation discipline

Glimpse of Generic Universe Types

```
class List<Y> {
   rep Node<Y> head;
   ...
}

class Node<X> {
   peer Node<X> next;
   ...
}
```



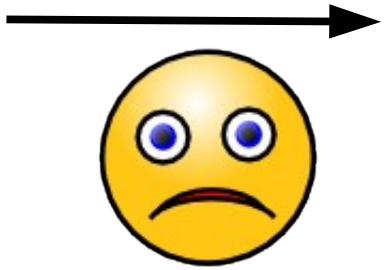
Generic Universe Types (GUT)

- Lightweight ownership type system
- Heap topology
- Owner-as-Modifier encapsulation discipline

- Large-scale use hampered by annotation effort
  - All fields, parameters, object creations, … need annotations

Manual annotation effort huge

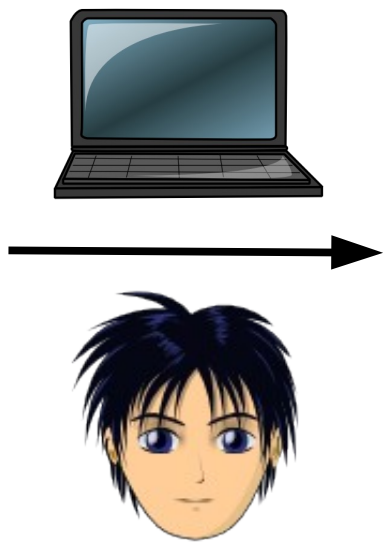
```
    AnnotatedTypeMirror lhsBase = lhs;
     while (lhsBase.getKind() != rhs.getKind()
            && (lhsBase.getKind() == TypeKind.WILDCARD || lhsBase.getKind() ==
TypeKind.TYPEVAR)) {
        if (lhsBase.getKind() == TypeKind.WILDCARD && rhs.getKind() !=
TypeKind.WILDCARD) {
            AnnotatedWildcardType wildcard = (AnnotatedWildcardType)lhsBase;
            if (lhsBase == null || !lhsBase.isAnnotated())
                return true;
            visited.add(lhsBase.getElement());
        } else if (rhs.getKind() == TypeKind.WILDCARD) {
            rhs = ((AnnotatedWildcardType)rhs).getExtendsBound();
        } else if (lhsBase.getKind() == TypeKind.TYPEVAR && rhs.getKind() !=
TypeKind.TYPEVAR) {
            AnnotatedTypeVariable lhsb_atv = (AnnotatedTypeVariable)lhsBase;
            Set<AnnotationMirror> lAnnos = lhsb_atv.getLowerBoundAnnotations();
            if (!lAnnos.isEmpty())
                return qualifierHierarchy.isSubtype(rhs.getAnnotations(), lAnnos);
rhs.getAnnotations().contains(qualifierHierarchy.getBottomQualifier());
        }
    }

    AnnotatedTypeMirror rhsBase = rhs.typeFactory.atypes.asSuper(rhs,
lhsBase);

    if (!qualifierHierarchy.isSubtype(rhsBase.getAnnotations(),
lhsBase.getAnnotations()))
        return false;

    if (lhs.getKind() == TypeKind.ARRAY && rhsBase.getKind() ==
TypeKind.ARRAY) {
        AnnotatedTypeMirror rhsComponent =
((AnnotatedArrayType)rhsBase).getComponentType();
        AnnotatedTypeMirror lhsComponent =
((AnnotatedArrayType)lhsBase).getComponentType();
        return isSubtypeAsArrayComponent(rhsComponent, lhsComponent);
    } else if (lhsBase.getKind() == TypeKind.DECLARED && rhsBase.getKind()
== TypeKind.DECLARED) {
        return isSubtypeTypeArguments((AnnotatedDeclaredType)rhsBase,
(AnnotatedDeclaredType)lhsBase);
    } else if (lhsBase.getKind() == TypeKind.TYPEVAR && rhsBase.getKind() ==
TypeKind.TYPEVAR) {
        AnnotatedTypeMirror rhsSuperClass = rhsBase;
        while (rhsSuperClass.getKind() == TypeKind.TYPEVAR) {
            rhsSuperClass = ((AnnotatedTypeVariable)
rhsSuperClass).getUpperBound();
        }
        Set<AnnotationMirror> las = ((AnnotatedTypeVariable)
lhsBase).getLowerBoundAnnotations();
        Set<AnnotationMirror> ras = ((AnnotatedTypeVariable)
rhsBase).getUpperBoundAnnotations();
        if (!las.isEmpty()) {
            return qualifierHierarchy.isSubtype(ras, las);
```



```
rep AnnotatedTypeMirror lhsBase = lhs;
     while (lhsBase.getKind() != rhs.getKind()
            && (lhsBase.getKind() == TypeKind.WILDCARD || lhsBase.getKind() ==
TypeKind.TYPEVAR)) {
        if (lhsBase.getKind() == TypeKind.WILDCARD && rhs.getKind() !=
TypeKind.WILDCARD) {

            rep AnnotatedWildcardType wildcard = (rep
AnnotatedWildcardType)lhsBase;  if (lhsBase == null || !lhsBase.isAnnotated())
                return true;
            visited.add(lhsBase.getElement());
        } else if (rhs.getKind() == TypeKind.WILDCARD) {

            rhs = ((peer AnnotatedWildcardType)rhs).getExtendsBound();
        } else if (lhsBase.getKind() == TypeKind.TYPEVAR && rhs.getKind() !=

TypeKind.TYPEVAR) { rep AnnotatedTypeVariable lhsb_atv = (rep
AnnotatedTypeVariable)lhsBase;

            rep Set<peer AnnotationMirror> lAnnos =
lhsb_atv.getLowerBoundAnnotations();
            if (!lAnnos.isEmpty())
                return qualifierHierarchy.isSubtype(rhs.getAnnotations(), lAnnos);
rhs.getAnnotations().contains(qualifierHierarchy.getBottomQualifier());
        }
    }

    } rep AnnotatedTypeMirror rhsBase =
rhs.typeFactory.atypes.asSuper(rhs, lhsBase);

    if (lhs.getKind() == TypeKind.ARRAY && rhsBase.getKind() ==
TypeKind.ARRAY) {

        peer AnnotatedTypeMirror rhsComponent = ((peer
AnnotatedArrayType)rhsBase).getComponentType();

        peer AnnotatedTypeMirror lhsComponent = ((peer
AnnotatedArrayType)lhsBase).getComponentType();
        return isSubtypeAsArrayComponent(rhsComponent, lhsComponent);
    } else if (lhsBase.getKind() == TypeKind.DECLARED && rhsBase.getKind()

== TypeKind.DECLARED) {rep AnnotatedTypeMirror rhsBase =
rhs.typeFactory.atypes.asSuper(rhs, lhsBase);
```

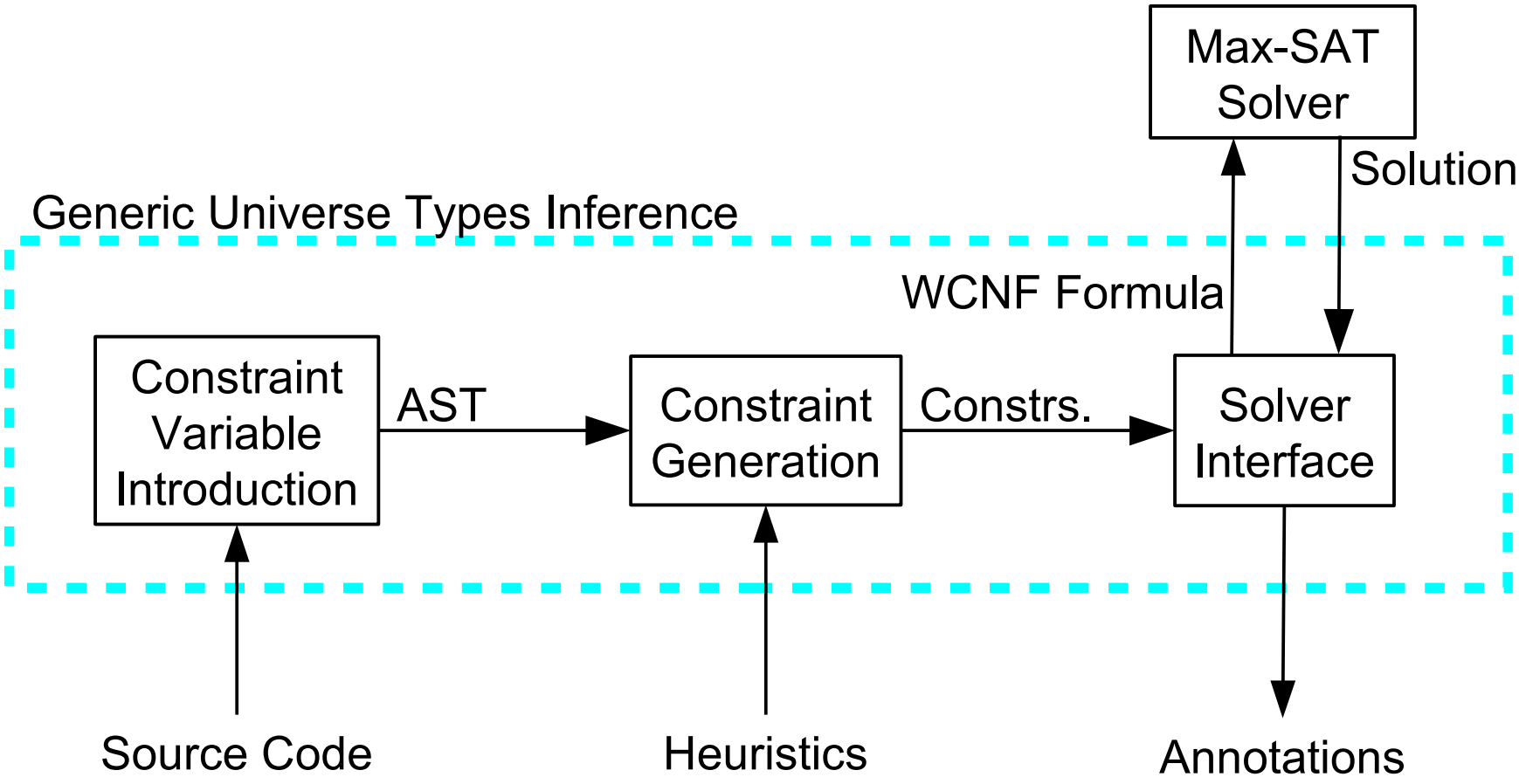
Automated annotation support

```
AnnotatedTypeMirror lhsBase = lhs;
 while (lhsBase.getKind() != rhs.getKind()
        && (lhsBase.getKind() == TypeKind.WILDCARD || lhsBase.getKind() ==
TypeKind.TYPEVAR)) {
      if (lhsBase.getKind() == TypeKind.WILDCARD && rhs.getKind() !=
TypeKind.WILDCARD) {
        AnnotatedWildcardType wildcard = (AnnotatedWildcardType)lhsBase;
        if (lhsBase == null || !lhsBase.isAnnotated())
          return true;
        visited.add(lhsBase.getElement());
      } else if (rhs.getKind() == TypeKind.WILDCARD) {
        rhs = ((AnnotatedWildcardType)rhs).getExtendsBound();
      } else if (lhsBase.getKind() == TypeKind.TYPEVAR && rhs.getKind() !=
TypeKind.TYPEVAR) {
        AnnotatedTypeVariable lhsb_atv = (AnnotatedTypeVariable)lhsBase;
        Set<AnnotationMirror> lAnnos = lhsb_atv.getLowerBoundAnnotations();
        if (!lAnnos.isEmpty())
          return qualifierHierarchy.isSubtype(rhs.getAnnotations(), lAnnos);
rhs.getAnnotations().contains(qualifierHierarchy.getBottomQualifier());
      }
    }

    AnnotatedTypeMirror rhsBase = rhs.typeFactory.atypes.asSuper(rhs,
lhsBase);

    if (!qualifierHierarchy.isSubtype(rhsBase.getAnnotations(),
lhsBase.getAnnotations()))
      return false;

    if (lhs.getKind() == TypeKind.ARRAY && rhsBase.getKind() ==
TypeKind.ARRAY) {
      AnnotatedTypeMirror rhsComponent =
((AnnotatedArrayType)rhsBase).getComponentType();
      AnnotatedTypeMirror lhsComponent =
((AnnotatedArrayType)lhsBase).getComponentType();
      return isSubtypeAsArrayComponent(rhsComponent, lhsComponent);
    } else if (lhsBase.getKind() == TypeKind.DECLARED && rhsBase.getKind()
== TypeKind.DECLARED) {
      return isSubtypeTypeArguments((AnnotatedDeclaredType)rhsBase,
(AnnotatedDeclaredType)lhsBase);
    } else if (lhsBase.getKind() == TypeKind.TYPEVAR && rhsBase.getKind() ==
TypeKind.TYPEVAR) {
      AnnotatedTypeMirror rhsSuperClass = rhsBase;
      while (rhsSuperClass.getKind() == TypeKind.TYPEVAR) {
        rhsSuperClass = ((AnnotatedTypeVariable)
rhsSuperClass).getUpperBound();
      }
      Set<AnnotationMirror> las = ((AnnotatedTypeVariable)
lhsBase).getLowerBoundAnnotations();
      Set<AnnotationMirror> ras = ((AnnotatedTypeVariable)
rhsBase).getUpperBoundAnnotations();
      if (!las.isEmpty()) {
        return qualifierHierarchy.isSubtype(ras, las);
```



```
rep AnnotatedTypeMirror lhsBase = lhs;
 while (lhsBase.getKind() != rhs.getKind()
        && (lhsBase.getKind() == TypeKind.WILDCARD || lhsBase.getKind() ==
TypeKind.TYPEVAR)) {
      if (lhsBase.getKind() == TypeKind.WILDCARD && rhs.getKind() !=
TypeKind.WILDCARD) {

        rep AnnotatedWildcardType wildcard = (rep
AnnotatedWildcardType)lhsBase;  if (lhsBase == null || !lhsBase.isAnnotated())
          return true;
        visited.add(lhsBase.getElement());
      } else if (rhs.getKind() == TypeKind.WILDCARD) {

        rhs = ((peer AnnotatedWildcardType)rhs).getExtendsBound();
      } else if (lhsBase.getKind() == TypeKind.TYPEVAR && rhs.getKind() !=

TypeKind.TYPEVAR) { rep AnnotatedTypeVariable lhsb_atv = (rep
AnnotatedTypeVariable)lhsBase;

        rep Set<peer AnnotationMirror> lAnnos =
lhsb_atv.getLowerBoundAnnotations();
        if (!lAnnos.isEmpty())
          return qualifierHierarchy.isSubtype(rhs.getAnnotations(), lAnnos);
rhs.getAnnotations().contains(qualifierHierarchy.getBottomQualifier());
      }

    } rep AnnotatedTypeMirror rhsBase =
rhs.typeFactory.atypes.asSuper(rhs, lhsBase);

    if (lhs.getKind() == TypeKind.ARRAY && rhsBase.getKind() ==
TypeKind.ARRAY) {

      peer AnnotatedTypeMirror rhsComponent = ((peer
AnnotatedArrayType)rhsBase).getComponentType();

      peer AnnotatedTypeMirror lhsComponent = ((peer
AnnotatedArrayType)lhsBase).getComponentType();
      return isSubtypeAsArrayComponent(rhsComponent, lhsComponent);
    } else if (lhsBase.getKind() == TypeKind.DECLARED && rhsBase.getKind()

== TypeKind.DECLARED) {rep AnnotatedTypeMirror rhsBase =
rhs.typeFactory.atypes.asSuper(rhs, lhsBase);
```

Architecture



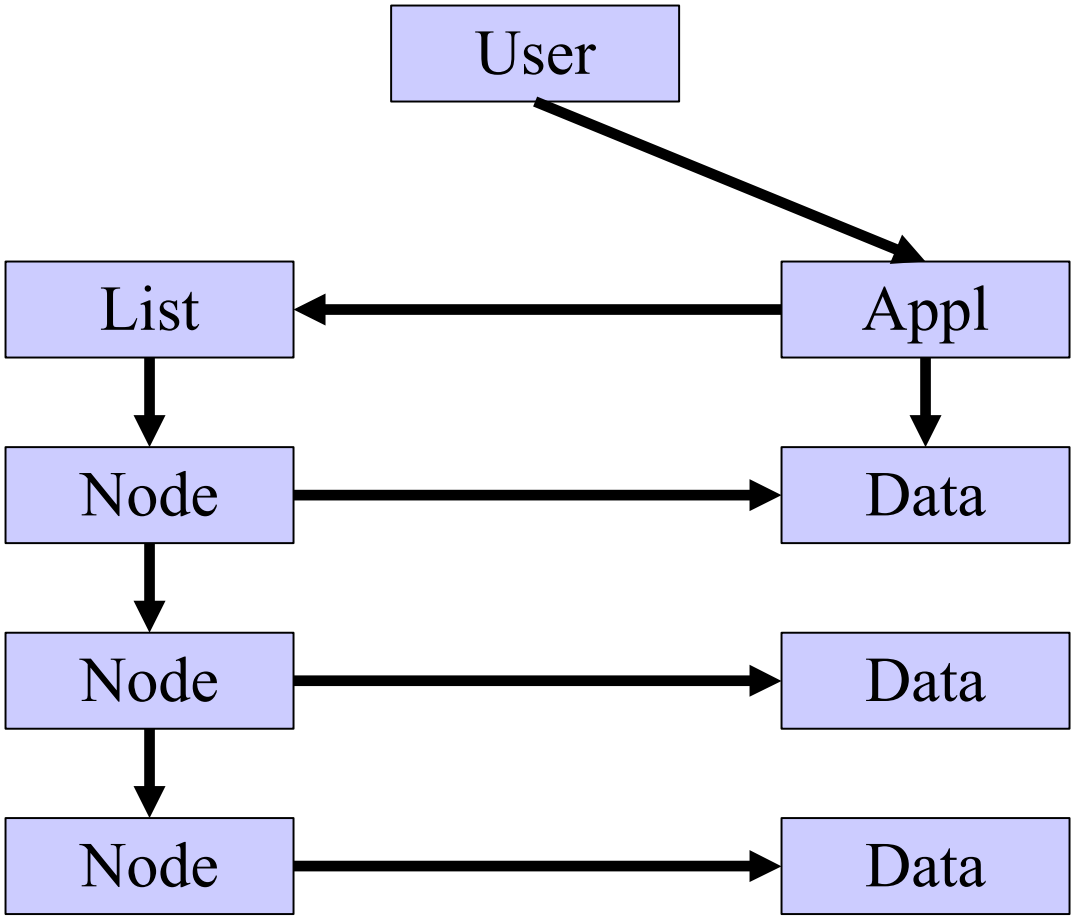
Many solutions exist

- Problem is different from usual type inference
- Not interested in only a typable solution
- We want a good structure

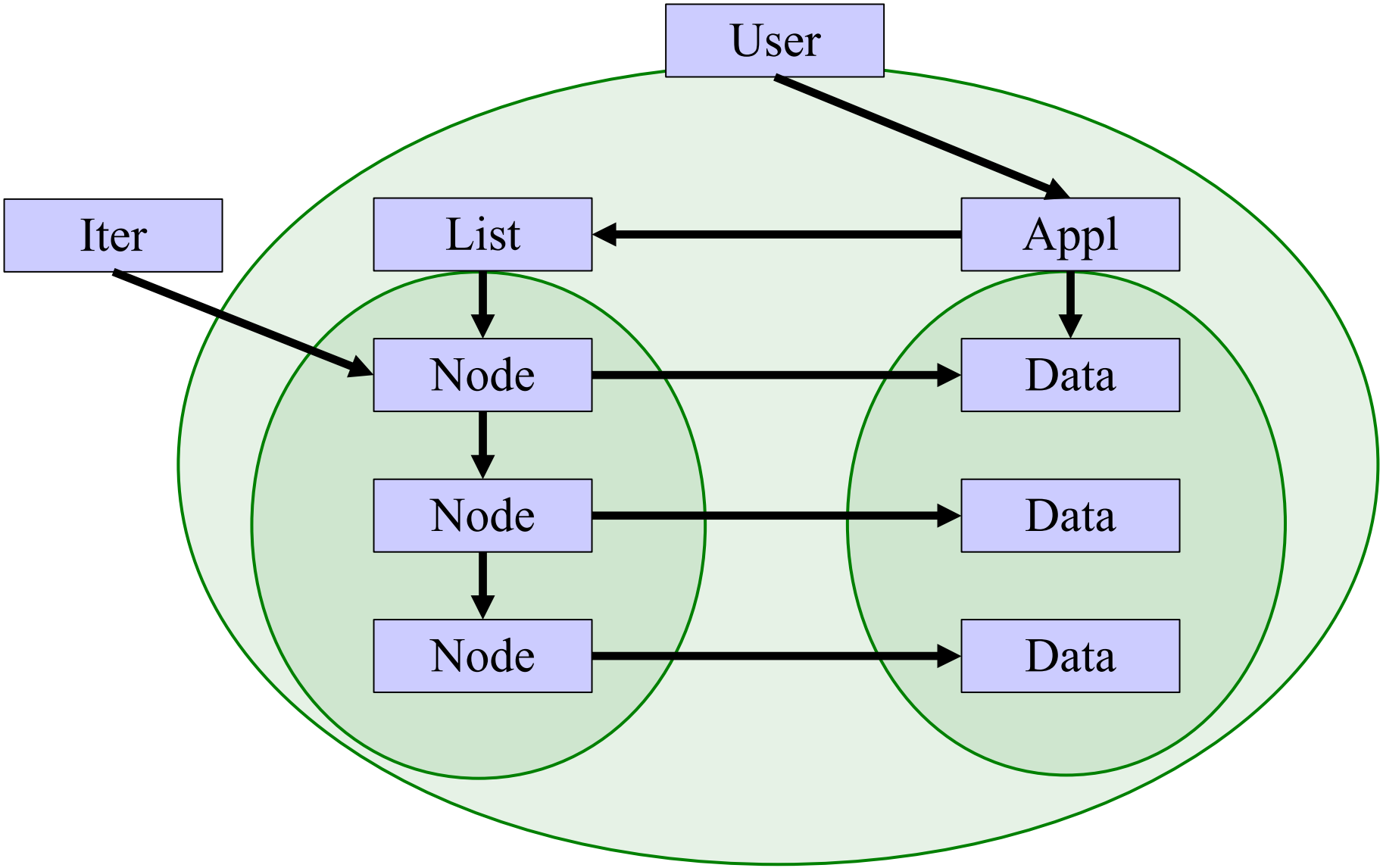
Outline

- Overview
- Tunable Static Inference for GUT
  - GUT motivation & example
  - Constraint variable introduction
  - Constraint generation
  - Max-SAT encoding
- Implementation & Evaluation
- Conclusion

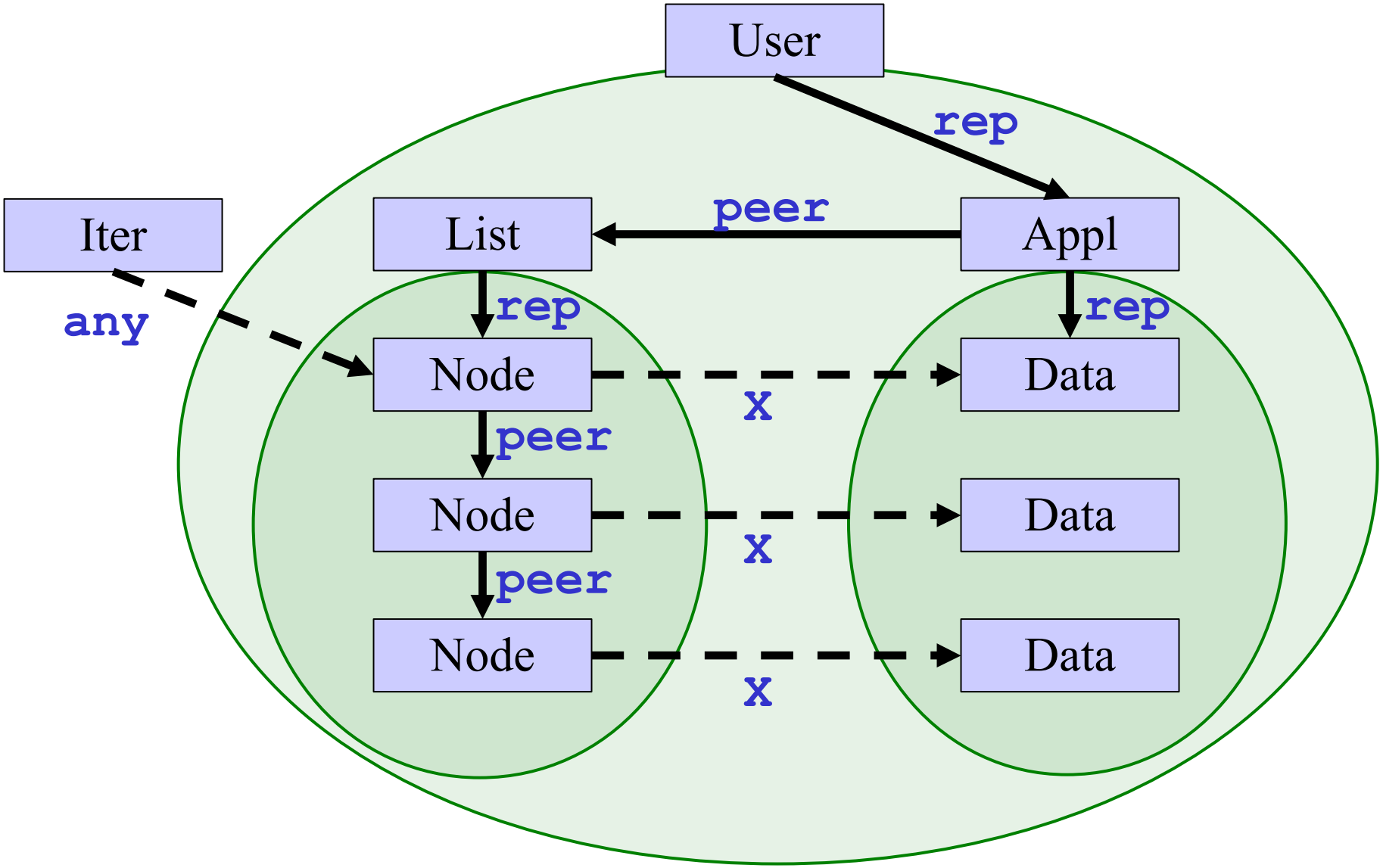
Intended Structure



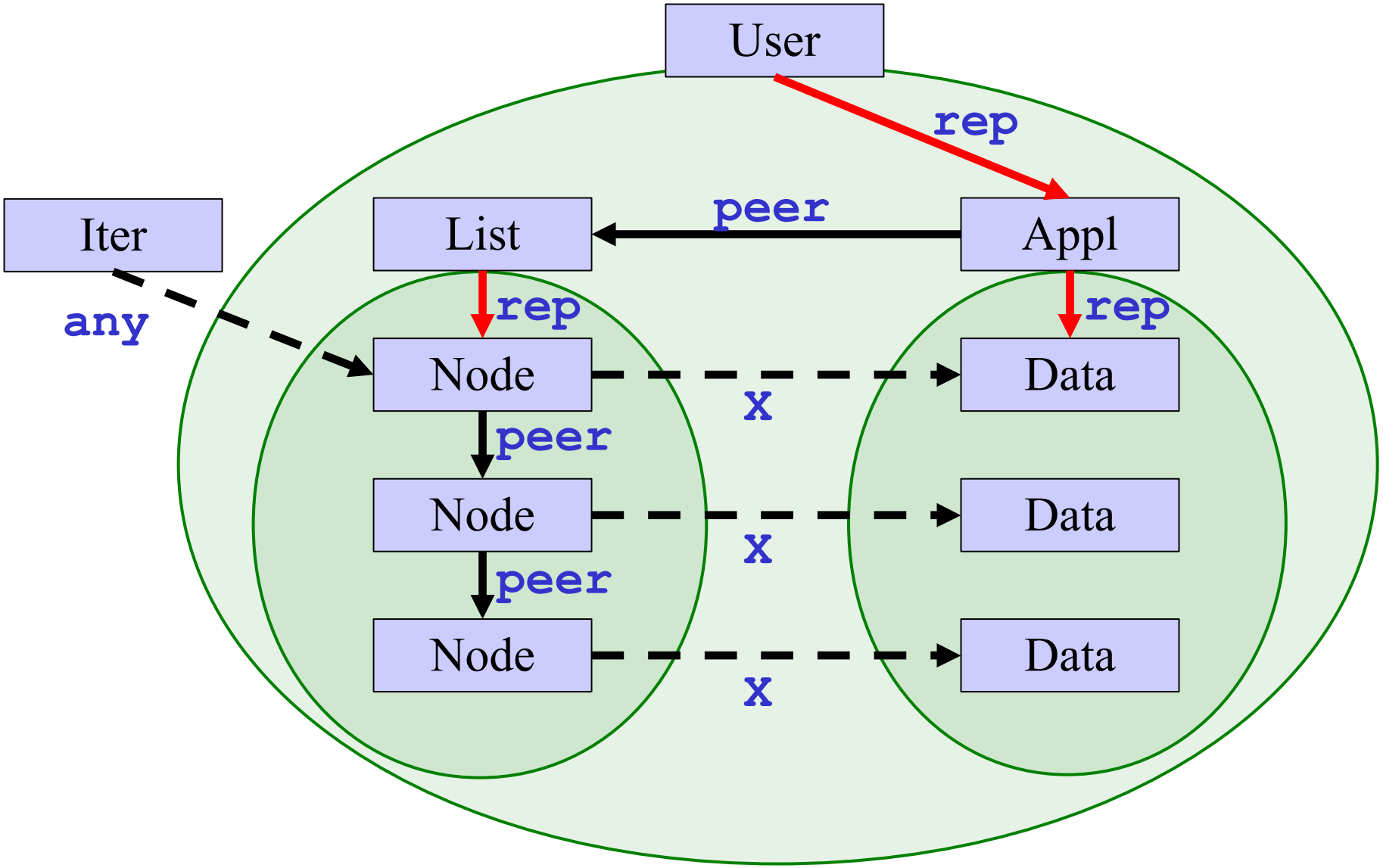
Object Ownership



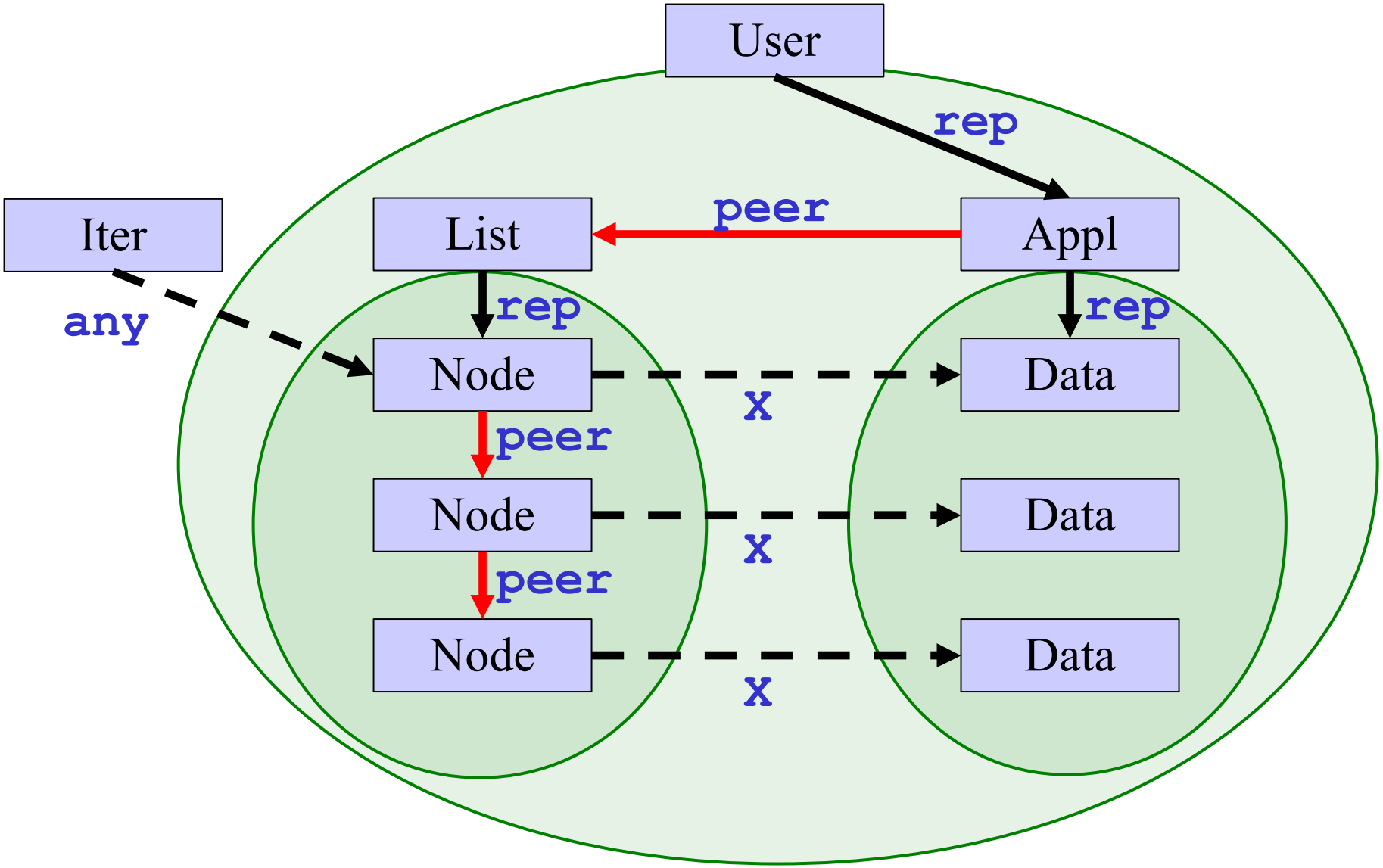
Generic Universe Types (GUT)



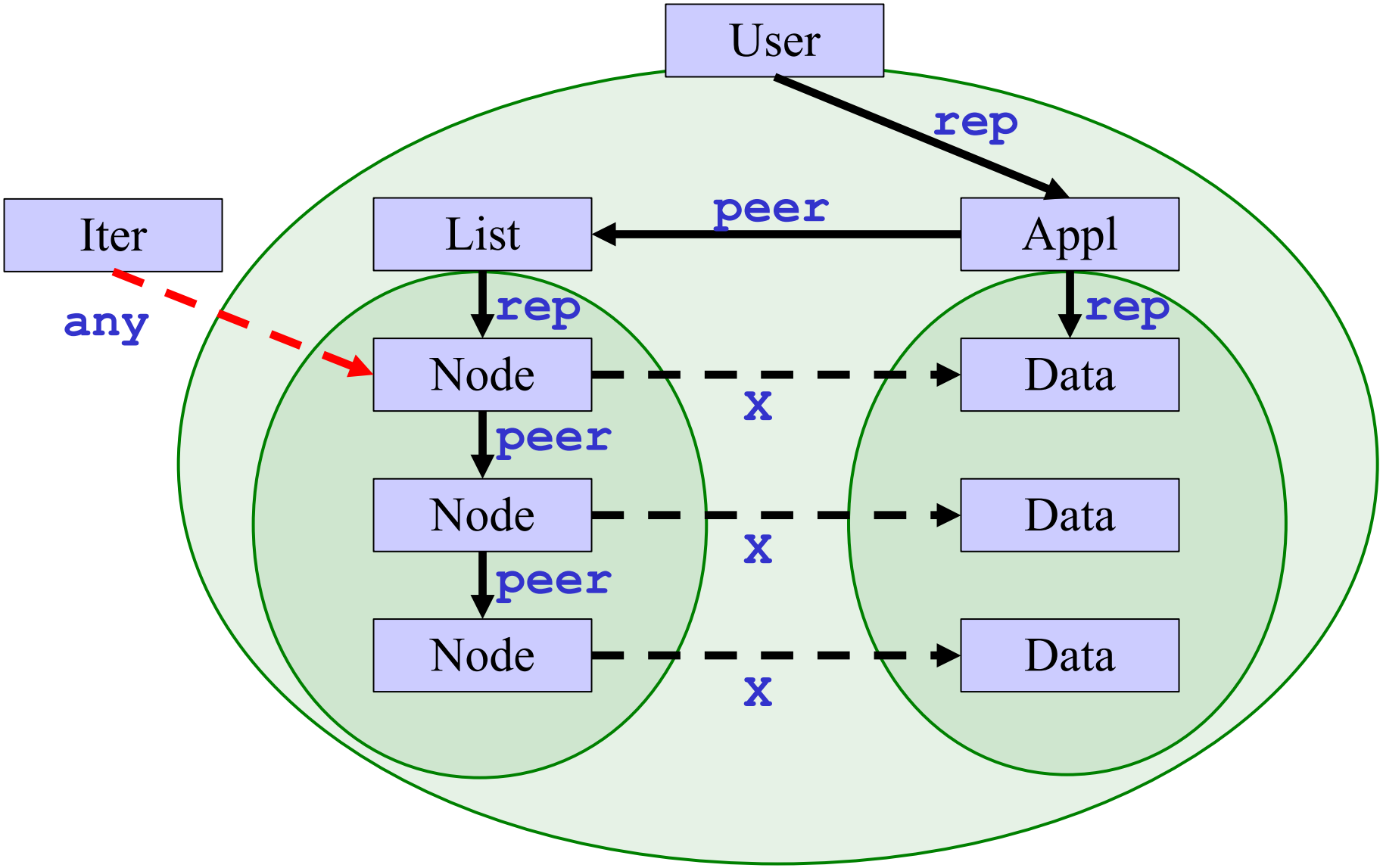
Generic Universe Types (GUT)



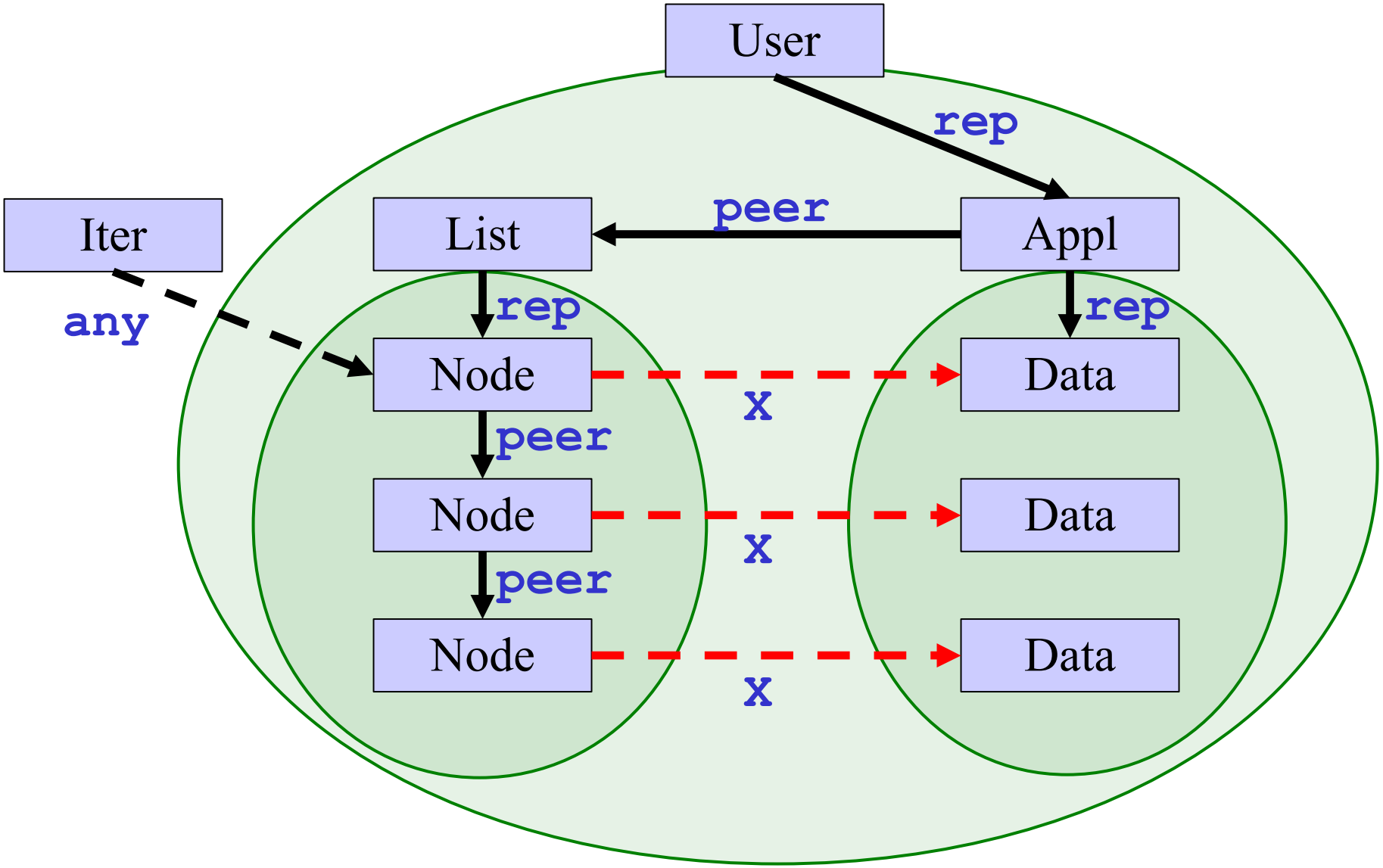
Generic Universe Types (GUT)



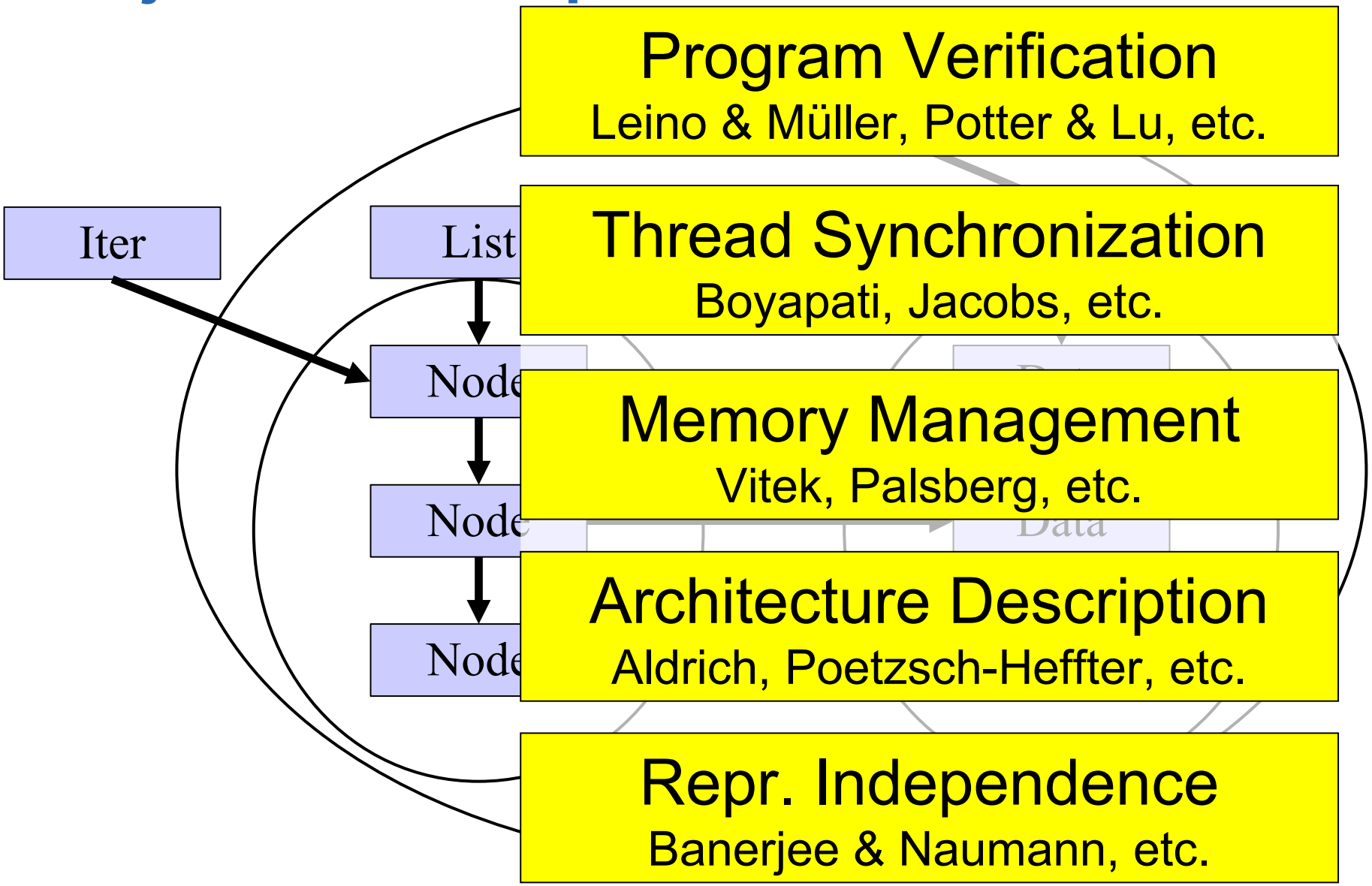
Generic Universe Types (GUT)



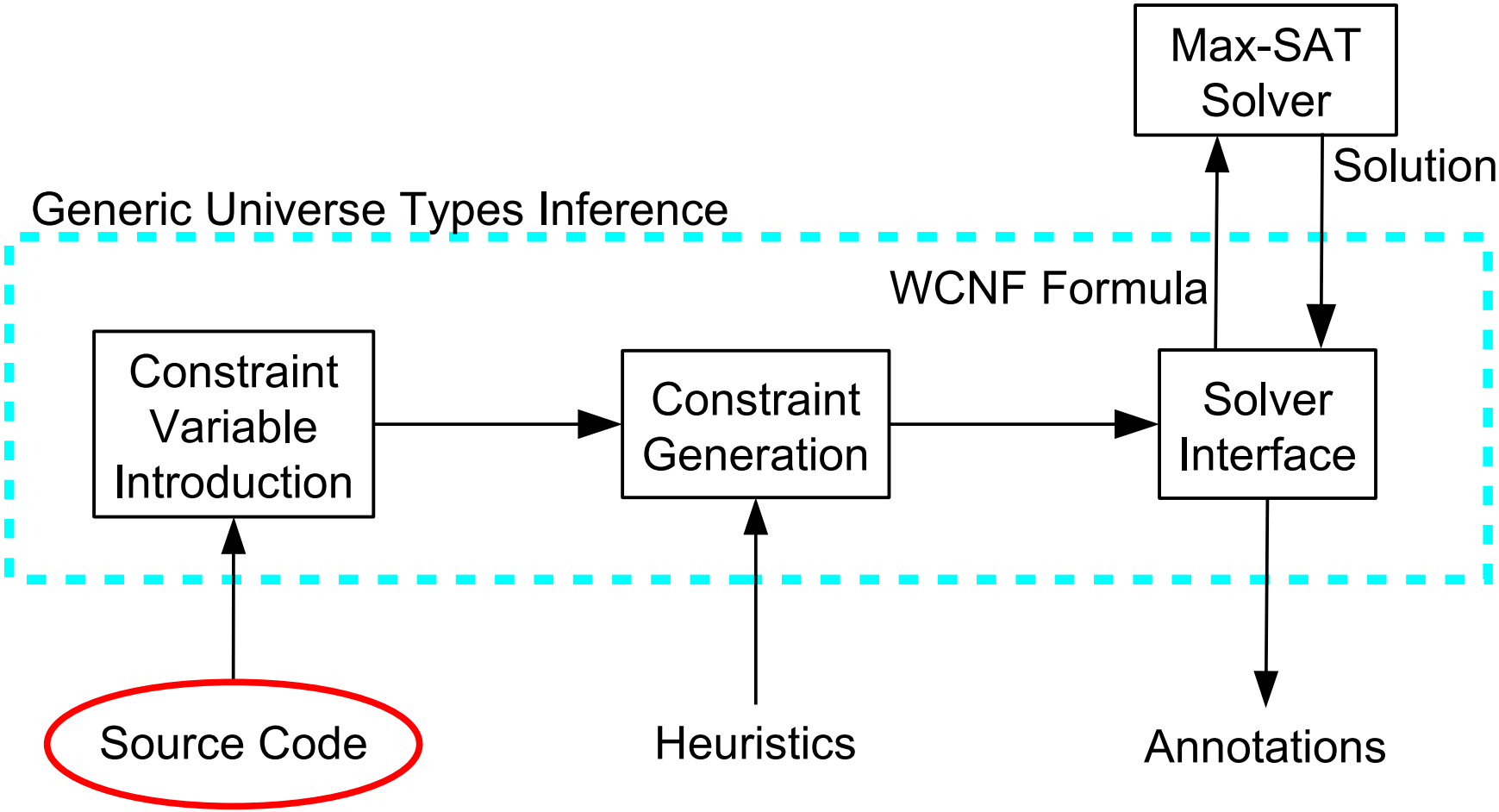
Generic Universe Types (GUT)



Object Ownership



Architecture



Programming Language

- Generic Featherweight Java
  - Extended with state and ownership
- Non-variable types:

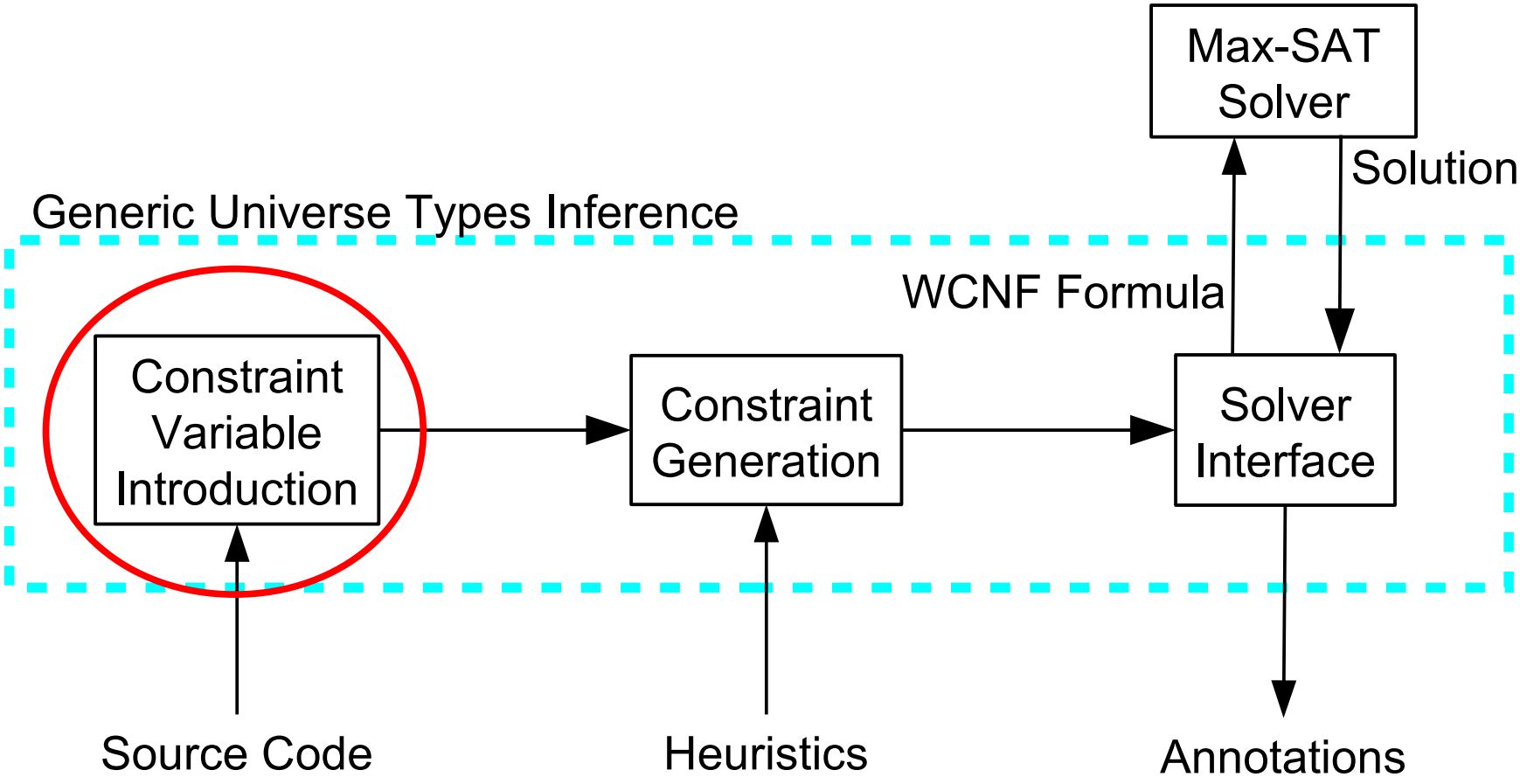
$$N \quad ::= \quad u\, C\langle \overline{T} \rangle$$

- Ownership modifiers:

$$u \quad ::= \quad \texttt{peer} \mid \texttt{rep} \mid \texttt{any} \mid \texttt{lost} \mid \texttt{self} \mid \alpha$$

Constraint Variables

Architecture



Constraint Variable Introduction

- Introduce constraint variables α for
  - Every reference type
  - Every expression

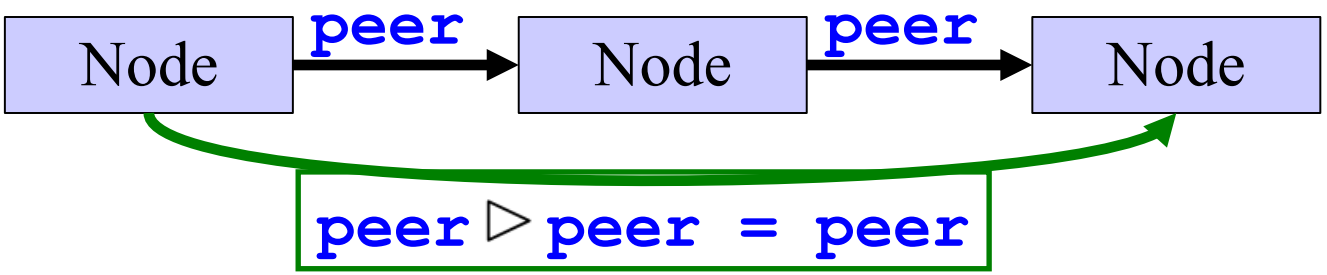
Generic Universe Types Example

```
class Node<X> {
  peer Node<X> next;
  X elem;
  void replaceNext(X p) { ...
    peer Node<X> tmp = next.next;
    ...
  }
}
class List<Y> {
  rep Node<Y> head;
  void dropFirst() {
    head = head.next;
  }
}
```

Adaptation of Ownership Modifiers

```
next.next
```

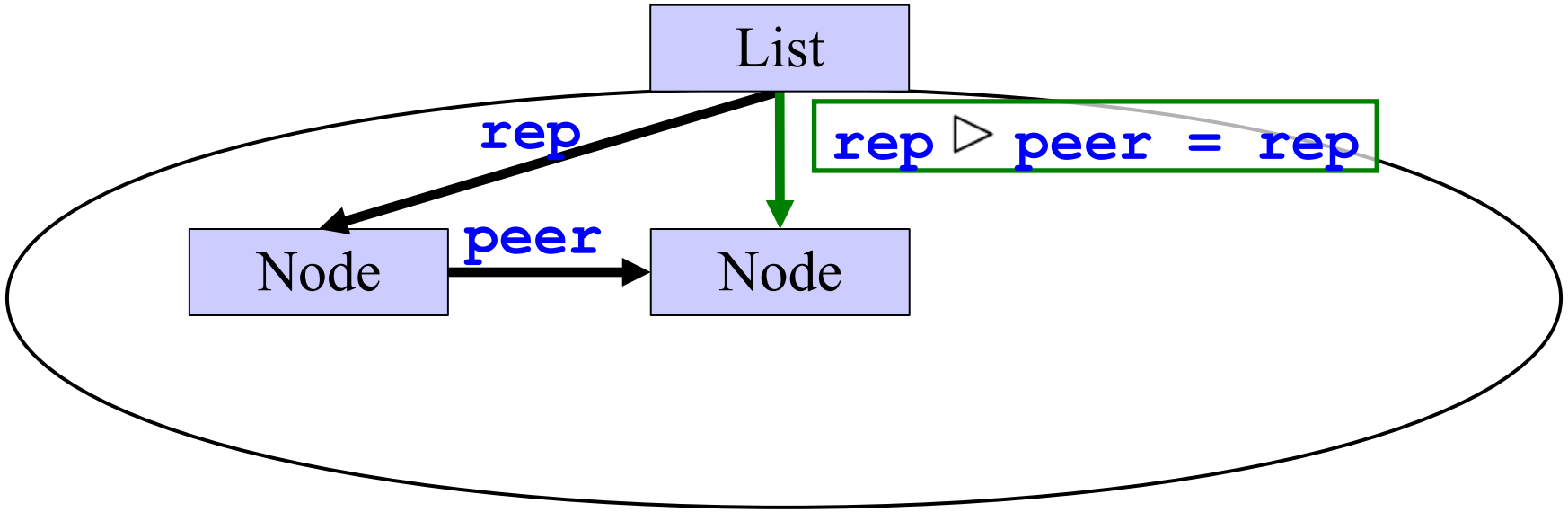
```
peer ▷ peer  = peer
```



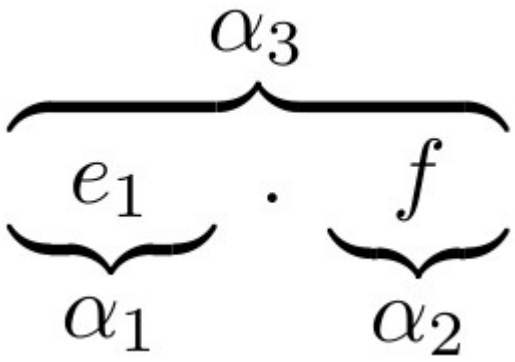
Adaptation of Ownership Modifiers

```
head.next
```

```
rep ▷ peer = rep
```

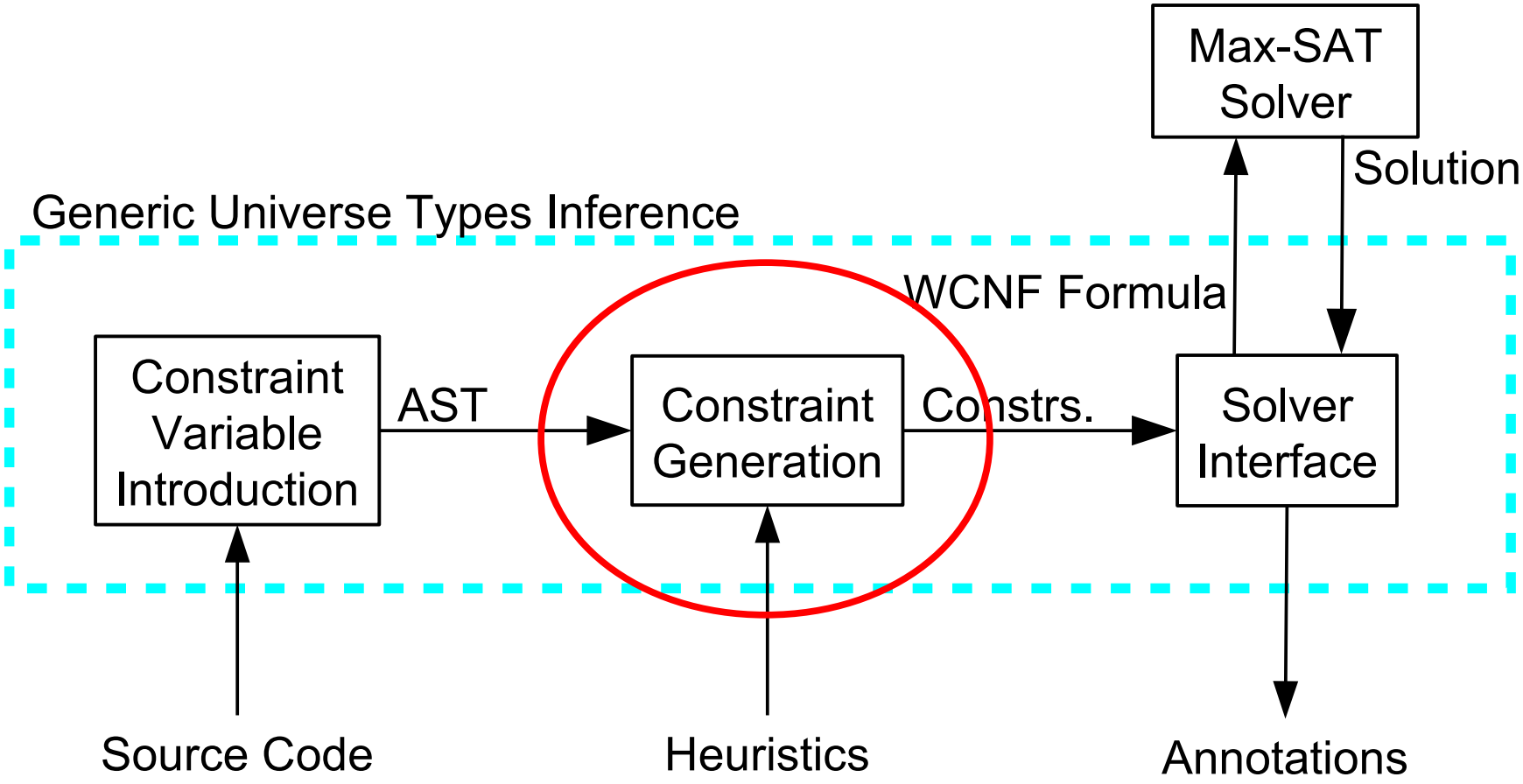


Adaptation of Ownership Modifiers



$$\alpha_1 \triangleright \alpha_2 = \alpha_3$$

Architecture



Constraints

- Subtype $u_1 <: u_2$
- Equality $u_1 = u_2$
- Inequality $u_1 \neq u_2$
- Comparable $u_1 <:> u_2$
- Adaptation $u_1 \triangleright u_2 = \alpha_3$

Constraint Generation

- Traverse AST and generate constraints corresponding to the GUT type rules
- Standard typing judgment

$$\Gamma \vdash e : T$$

- Constraint generation rule

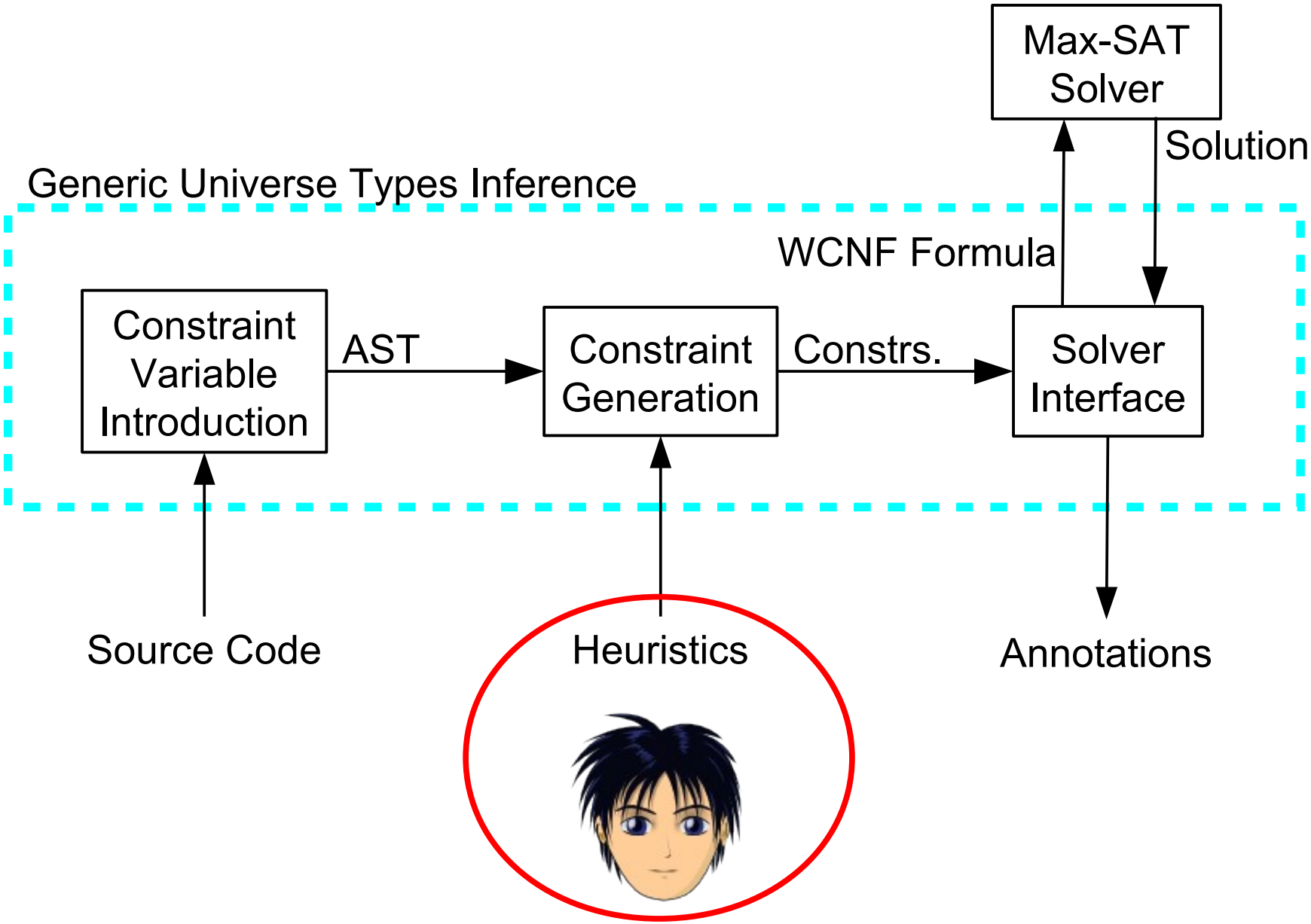
$$\Gamma \vdash e : T, \Sigma$$

Constraint Set

Constraint Generation – Field Update

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : N_0, \Sigma_0 \\ \Gamma \vdash e_1 : T_1, \Sigma_1 \\ \mathrm{fType}(N_0, f) = T_2, \Sigma_2 \\ \Gamma \vdash T_1 <: T_2 : \Sigma_3 \\ \Sigma_4 = \{\texttt{lost} \notin T_2\}\end{array}}{\Gamma \vdash e_0.f := e_1 : T_2, \bigcup_{i=0}^{i=4} \Sigma_i}$$

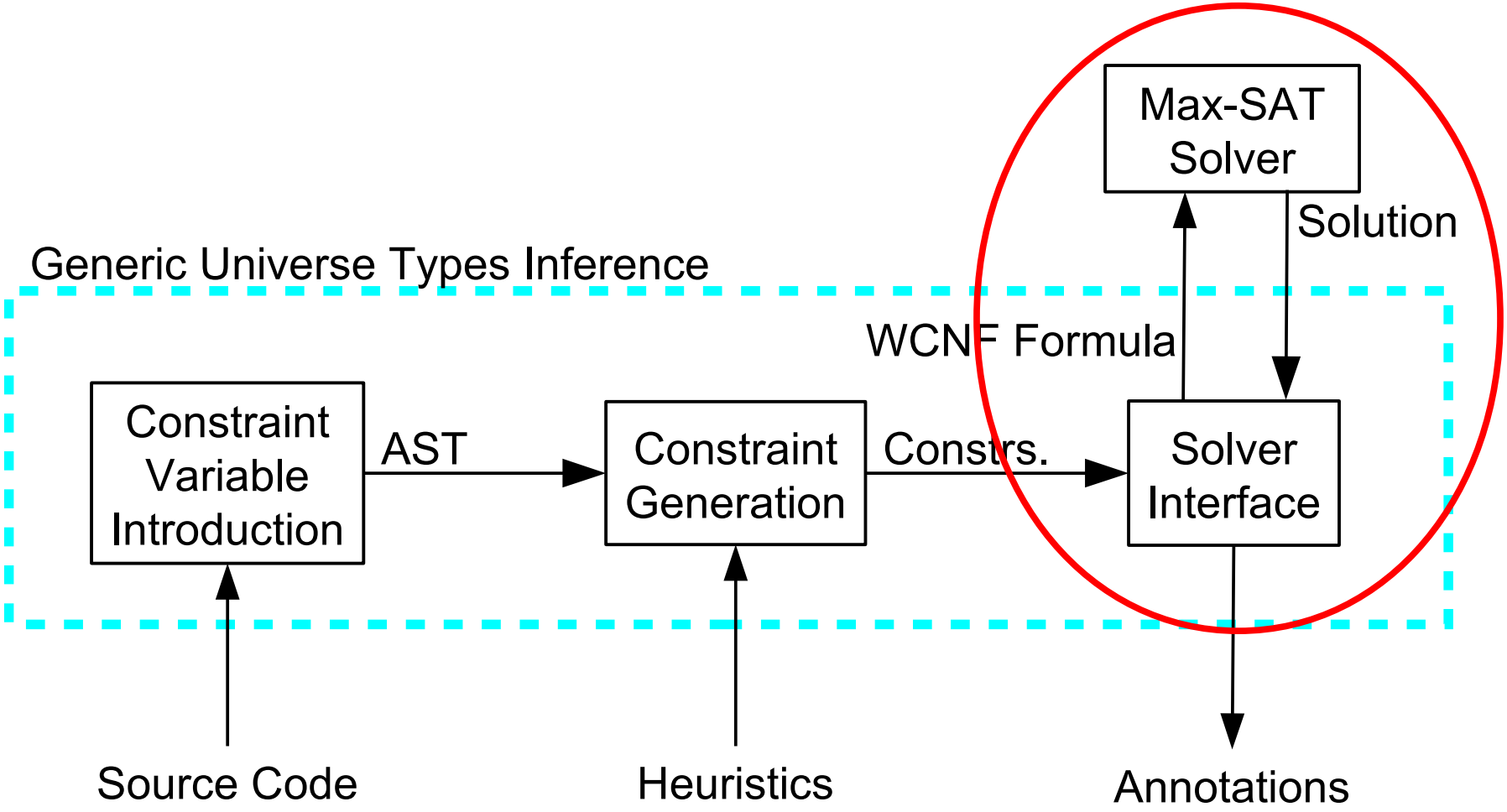
Architecture



Multiple solutions are possible

- Solving the constraints might give flat structure
- Add new, optional and weighted constraints to encode preferences
- Goal: satisfy as many preferences as possible
- For constraint variable α that appears in
  - Field types, prefer **`rep`**
  - Return types, prefer **`rep`**
  - Parameter types, prefer **`any`**
  - Type variable bounds, prefer **`any`**

Architecture



Boolean representation

- Each constraint variable encoded by four booleans

$$\alpha \Longleftrightarrow \beta^{peer}, \beta^{rep}, \beta^{any}, \beta^{lost}$$

Converting constraints to CNF formulas

Constraint	Encoding
$\alpha_1 <: \alpha_2$	$(\beta_1^{any} \Rightarrow \beta_2^{any}) \wedge (\beta_2^{peer} \Rightarrow \beta_1^{peer}) \wedge$ $(\beta_2^{rep} \Rightarrow \beta_1^{rep}) \wedge (\beta_1^{lost} \Rightarrow (\beta_2^{lost} \vee \beta_2^{any}))$
$\alpha_1 = \alpha_2$	$(\beta_1^{peer} \Rightarrow \beta_2^{peer}) \wedge (\beta_1^{rep} \Rightarrow \beta_2^{rep}) \wedge$ $(\beta_1^{lost} \Rightarrow \beta_2^{lost}) \wedge (\beta_1^{any} \Rightarrow \beta_2^{any})$
$\alpha_1 \neq \alpha_2$	$(\beta_1^{peer} \Rightarrow \neg\beta_2^{peer}) \wedge (\beta_1^{rep} \Rightarrow \neg\beta_2^{rep}) \wedge$ $(\beta_1^{lost} \Rightarrow \neg\beta_2^{lost}) \wedge (\beta_1^{any} \Rightarrow \neg\beta_2^{any})$
$\alpha_1 <:> \alpha_2$	$(\beta_1^{peer} \Rightarrow \neg\beta_2^{rep}) \wedge (\beta_1^{rep} \Rightarrow \neg\beta_2^{peer})$
$\alpha_1 \rhd \alpha_2 = \alpha_3$	$(\beta_1^{peer} \wedge \beta_2^{peer} \Rightarrow \beta_3^{peer}) \wedge$ $(\beta_1^{rep} \wedge \beta_2^{peer} \Rightarrow \beta_3^{rep}) \wedge$ $(\beta_2^{any} \Rightarrow \beta_3^{any}) \wedge (\beta_2^{lost} \Rightarrow \beta_3^{lost}) \wedge$ $(\beta_1^{any} \wedge \neg\beta_2^{any} \Rightarrow \beta_3^{lost}) \wedge$ $(\beta_1^{lost} \wedge \neg\beta_2^{any} \Rightarrow \beta_3^{lost}) \wedge (\beta_2^{rep} \Rightarrow \beta_3^{lost})$

Outline

- Overview
- Tunable Static Inference for GUT
  - GUT motivation & example
  - Constraint variable introduction
  - Constraint generation
  - Max-SAT encoding
- Implementation & Evaluation
- Conclusion

Implementation

- Built on top of the OpenJDK Java compiler
- Written in Scala
- Result in Annotation File Utilities format
- Type checker for GUT
  - Uses JSR 308 type annotation syntax `@Peer`

Evaluation

Benchmark	SLOC	Timing	peer	rep	any
zip	2611	5.6s	67%	18%	15%
javad	1846	4.5s	51%	24%	25%
jdepend	2460	6.5s	64%	21%	15%
classycle	4658	7.8s	73%	13%	14%

Of the annotations inserted into the source code (excluding viewpoint adaptation)

Correct 

Desirable 

Related Work

- Milanova et al. (TOOLS 2011, IWACO 2011)
  - Static dominance inference on alias graphs
  - Only partial annotations
- Beckman & Nori (PLDI 2011)
  - Typestate system
  - Probabilistic constraints allow overconstrained systems
- Welsch & Schäfer (TOOLS 2011)
  - Location type system
  - IDE integration and overconstrained systems

Future Work

Generalized inference framework

- Other ownership type systems
- Other type systems
- Other solvers

Tunable Static Inference for GUT

- Infers ownership type annotations
- Preferences among multiple legal typings
- Uses a Max-SAT solver as back-end
- Gives correct and desirable annotations
- Tool available from:

http://www.cs.washington.edu/homes/wmdietl/

http://checker-framework.googlecode.com/