Converting Java Classes to Use Generics [*]

Daniel von Dincklage
Department of Computer Science
University of Colorado
danielvd@cs.colorado.edu

Amer Diwan
Department of Computer Science
University of Colorado
diwan@cs.colorado.edu

ABSTRACT

Generics offer significant software engineering benefits since they provide code reuse without compromising type safety. Thus generics will be added to the Java language in the next release. While this extension to Java will help programmers when they are writing new code, it will not help legacy code unless it is rewritten to use generics. In our experience, manually modifying existing programs to use generics is complex and can be error prone and labor intensive.

We describe a system, Ilwith, that (i) converts non-generic classes to generic classes and (ii) rewrites their clients to use the newly generified classes. Our experiments with a number of Java container classes show that our system is effective in modifying legacy code to use generics.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Languages

Keywords

Generics, Parametric Polymorphism, Type Inference

1. INTRODUCTION

Generics, a special case of bounded parametric polymorphism [3], are useful for writing data-structures that are reusable without compromising static type checking. For example, a generic list package, parameterized by its element type, is not only reusable, but also preserves type information via the type parameter. In contrast, a non-generic list package, as distributed with Java [8], is reusable but operations on a list return or take parameters of type `Object`. Thus clients of the list must explicitly apply narrowing conversions, compromising static checking and readability of code. In addition to the software engineering benefits, generics can also provide performance benefits. For example, the increased type information provided by generics can give compilers more freedom in reordering computations [4, 13]. It is therefore no surprise that many modern languages such as Eiffel [10], Modula-3 [12], and SML [11] support generics. Java does not yet support generics but will do so with the next major release.

In order to get the most benefit from the addition of generics to Java, programmers will need to modify existing classes. We show that even for simple classes this process is complex and thus can be error prone. To help programmers in rewriting existing classes to use generics we describe and evaluate a system, Ilwith, that performs two functions: (i) it analyzes existing Java classes and discovers opportunities for converting them into generic classes; (ii) it analyzes clients of non-generic classes and discovers how to modify the clients to use generic versions of the classes. Programmers may use the output of Ilwith as a guideline for manually rewriting classes to use generics or may feed the output into a tool that performs the rewriting automatically.

Ilwith analyzes the classes it should generify, deriving constraints on the types of all variables (instance variables, local variables, parameters, etc.). These constraints are subset based [7] and capture all uses and definitions of variables in the analyzed classes. They determine the range of types one can use for each variable without violating the type safety of the analyzed classes. Ilwith uses these constraints to identify and bound type parameters for the generic classes and to instantiate any generic classes used by the classes being analyzed.

While the constraint-based analysis described above produces Java programs that use generics and are correct with respect to the Java type system, it is too aggressive: it results in generic classes with a large number of type parameters even for simple classes. In the worst case, the approach above can result in generic classes with type parameters for each variable, method parameter, return type, etc. While these parameters are all correct, they are not what a programmer would have chosen: programmers tend to pick one or a few type parameters for each generic class, not tens or hundreds of parameters. Thus, to reduce the number of type

[*]This work was supported by NSF ITR grant CCR-0085792, an NSF Career Award CCR-0133457, and an IBM faculty partnership award. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

Listing 1: Dependency between the two tasks

```
class ListWrapper {
  void set( Object arg ) {
    data.add(arg);
  }

  Object get() {
    return data.get(0);
  }

  List data;
}
```

Listing 2: A simple example

```
class Example {
  void set( Object o ) {
    data = o;
  }

  Object get() {
    return data;
  }

  Object data;
}
```

parameters, Ilwith employs a number of coarsening heuristics. We designed our initial set of heuristics based on our study of how Eiffel programs use generics. However, the set of heuristics is extensible and we expect to revise them as we gain experience with our system.

Ilwith is modular: it does not need whole program analysis. For *best* results, Ilwith needs to analyze a class after all classes that it depends on have already been converted to use generics. In the case of cyclic dependencies, Ilwith can analyze multiple classes at once.

Our experience with Ilwith indicates that it often produces the same output as an expert programmer. When Ilwith was unable to produce the same output as our manual effort, we found the cause to be type problems in the Java class that were masked by weak type checking due to a lack of generics.

The remainder of this paper is organized as follows: Section 2 describes our approach intuitively. Section 3 gives the language that Ilwith operates upon and its type system. Section 4 presents the basic algorithm used in Ilwith. Section 5 refines our basic algorithm to handle Java-specific features. Section 6 presents and motivates heuristics for improving the quality of results produced by Ilwith. Section 7 presents our results. Section 8 presents related work and Section 9 concludes.

2. INTUITION BEHIND OUR APPROACH

In order to convert classes to take advantage of generics, Ilwith must perform two tasks: (i) convert non-generic classes to generic classes, which involves determining what types to make into formal generic parameters and the constraints to use for the parameters; (ii) convert clients of a non-generic class to use the generic version of the class.

While, for ease of explanation, we will treat these tasks as independent, we must actually perform them simultaneously to achieve the best results. For example, consider the class `ListWrapper` (Listing 1) which is a simple wrapper around the class `List`. Assume that `List` is a generic class that takes a single type parameter determining the type of its elements. If we determine the type formals for a generic `ListWrapper` without considering the type parameters to `List`, we will have to assume that `data.add` requires an `Object` and `data.get` returns an `Object`. On the other hand, if we determine the type actuals to `List` without simultaneously converting `ListWrapper` to a generic class, we will end up instantiating `List` with type `Object`. In both cases we will fail to convert `ListWrapper` into a generic class.

We now use the class in Listing 2 to intuitively describe

Listing 3: With replaced type variables

```
class Example {
  void set( τ1 o ) {
    data = o;
  }

  τ2 get() {
    return data;
  }

  τ3 data;
}
```

our algorithm. Section 4 describes the algorithm more formally.

Ilwith starts by replacing all explicit type references with fresh type variables (Listing 3). Next, Ilwith generates constraints on the type variables based on the body of the class. For our example, Ilwith produces two constraints: (i) τ_1, is a subtype or equal to τ_3 from the statement `data = o` and (ii) τ_3 is a subtype or equal to τ_2 from the statement `return data`.

Since generic Java allows type formals to be bound only from above (i.e., only by supertype constraints), τ_1 is the only variable that Ilwith can turn into a type parameter for the generic `Example` class (both τ_2 and τ_3 are bound from below). Ilwith next looks at the constraints on τ_1 to determine what bound to use for it. Since there are no non-trivial constraints on τ_1 (it is constrained from above only by type variables, which Ilwith can fix after it has a type for τ_1) Ilwith makes τ_1 into a type formal bound by `Object`. Finally, Ilwith picks the types to use for τ_2 and τ_3. τ_2 and τ_3 are constrained from below by τ_1 (which has been picked to be a type parameter) and from above by `Object` (from their declarations). Thus any type that is a supertype of τ_1 and a subtype of `Object` will be a legal instantiation for τ_2 and τ_3. Since we have already decided not to make τ_2 and τ_3 into type parameters, we pick `Object` instead of τ_1. Listing 4 shows the output of Ilwith when using this approach.

While the code in Listing 4 is a generic version of the class in Listing 2, it is not what an expert programmer would produce. The problem here is the opposite of what program analyses typically face: the output of the analysis is "too precise" rather than "not precise enough". By introducing subtype constraints for each assignment, Ilwith assumes that it is desirable to exploit subtype polymorphism whenever possible. Ilwith addresses this using heuristics that selectively introduce imprecision into the analysis. Ilwith is

Listing 4: The output with the basic algorithm

```
class Example⟨ τ <: Object ⟩ {
  void set( τ o ) {
    data = o;
  }

  Object get() {
    return data;
  }

  Object data;
}
```

Listing 5: The desired output

```
class Example⟨ τ <: Object ⟩ {
  void set( τ o ) {
    data = o;
  }

  τ get() {
    return data;
  }

  τ data;
}
```

extensible with respect to heuristics; the initial set of heuristics capture our intuition for styles that result in a good generic class.

By applying a heuristic (Section 6.2) designed to limit the overzealous use of subtype polymorphism, Ilwith is able to merge τ_1, τ_2 and τ_3 into a single type variable. After choosing this new type variable as a formal generic parameter and calculating the bound, Ilwith produces the desired output, as shown in Listing 5.

3. ANALYZED LANGUAGE AND NOTATION

This section describes the language that we operate on, the type system that we use, and notation in the rest of the paper.

3.1 Analyzed Language

Ilwith operates on a simple representation of Java programs that is at the same level as Java bytecodes except that it is expression based (as opposed to stack based). Our representation simplifies the task of our analysis in the following ways: (i) It makes all pointer dereferences (including accesses of instance variables of "this") explicit; (ii) It breaks down multi-level pointer dereferences into a sequence of single-level dereferences; (iii) It rewrites statements so that each statement has at most one dereference; (iv) It renames variables to be globally unique; (v) It introduces variables to represent the return value of each method; (vi) It resolves overloading.

3.2 Type Variables

Our algorithm starts by substituting a type variable for every Java type in the program. Ilwith uses two forms of type variables. Since the form of the type variables affects the precision of our analysis we describe them in some detail here.

We use $\tau_{(x,V)}$ to represent the type parameter that will be passed to `V` in the declaration of `x`. For example, let's suppose `x` is declared to be of type `List`$\langle\ldots\rangle$ and the generic type `List` takes a single type parameter called `E`. Then $\tau_{(x,E)}$ is the type parameter passed to `List` in the declaration of `x`.

The above notation is useful if we already know that a type is generic (e.g., `List` is generic in the example above). When Ilwith is analyzing mutually dependent classes, it may not know that a class is generic until late in the analysis. Thus, we need the second representation. $\tau_{(x,[\![v]\!])}$ represents the type of `v`, when accessed though `x`, for example `x.v`. $[\![v]\!]$ maps v to the equivalence class that represents v. An equivalence class represents all variables that are merged together (Section 6.2). Since the $[\![v]\!]$ is rarely needed, we will often use $\tau_{(x,v)}$ as an abbreviation for $\tau_{(x,[\![v]\!])}$. Also, for local and instance variables of the class being analyzed, we will often use the abbreviation $\tau_{(a)}$ instead of $\tau_{(this,a)}$. We treat local variables as instance variables.

Our implementation actually uses a slight variation of the above. To see why, consider the expressions `x.v.a` and `y.v.a`. Our type names can represent only one level of dereference (e.g., $\tau_{(v,a)}$) and thus would not be able to distinguish between the types of `x.v.a` and `y.v.a`. Our implementation essentially chains type variables to precisely represent the type of variables reached with a long access path.

3.3 Types Used by our Analysis

Our analysis needs to be able to find the unique lowest-common supertype and unique greatest-common subtype of two types. In Java, due to interfaces, it is not always possible to determine the non-trivial (i.e., not `Object`) lowest-common supertype of two types. Thus, we use a variant of the Java type system in which any two types have a unique lowest-common supertype and greatest common subtype. Ilwith's types include a powerset of all Java classes in the classes being analyzed.

The mapping from a Java type, $\mathtt{T}_1$ to an Ilwith type is the smallest set of types that includes $\mathtt{T}_1$ and all its supertypes in the Java type system. As the elements in Ilwith's type lattice are sets, the least-common supertype operation is the intersection of the two sets; the greatest-common subtype the union of the two sets. It should be clear that there is always a unique lowest-common supertype and greatest-common subtype in this type system. Donovan *et al.* [5] use a similar type system in their tool, using the term *union type* for a set of types.

To see the benefit of Ilwith's type system, let's suppose Ilwith needs to compute the least-common supertype of two Java classes $\mathtt{C}_1$ and $\mathtt{C}_2$, both of which implement the interfaces $\mathtt{I}_1$ and $\mathtt{I}_2$. Since there are two common immediate supertypes of $\mathtt{C}_1$ and $\mathtt{C}_2$, there is no unique least-common supertype. In Ilwith's type system, the $\mathtt{C}_1$ and $\mathtt{C}_1$ would be represented by $\{\mathtt{C}_1, \mathtt{I}_1, \mathtt{I}_2, \mathtt{Object}\}$ and $\{\mathtt{C}_2, \mathtt{I}_1, \mathtt{I}_2, \mathtt{Object}\}$ respectively. In this type system, we can easily compute the least-common supertype of these types: it is simply the intersection of the two sets (i.e., $\{\mathtt{I}_1, \mathtt{I}_2, \mathtt{Object}\}$).

Ultimately Ilwith must map its types to the Java types. However, there is no guarantee that the Ilwith type will actually exist in the original Java classes being analyzed. If such a type does not exist, then Ilwith can either choose an alternative "real" class, e.g., `Object` (thereby incurring a loss of precision) or ask the user for help (e.g., the user can create a new type).

If a Ilwith type contains a type that has been made generic, the conversion to a Java type is more complicated. Let's suppose Ilwith needs to convert the type of the type variable $\tau_{(a)}$ – {Object, Set, SortedSet} – to a Java type. Assume that SortedType and Set are generic: Set has a formal generic parameter F1, SortedSet has a formal generic parameter F2 and extends Set⟨F2⟩. Since SortedSet is the common subtype of all types in the type set {Object, Set, SortedSet}, Ilwith picks SortedSet. To instantiate SortedSet Ilwith takes the least common subtype of the parameters passed to Set and SortedSet in the declarations of the variables represented by $\tau_{(a)}$ (i.e., $\tau_{(a,F1)}$ and $\tau_{(a,F2)}$). In this case it is valid to intersect the types computed for $\tau_{(a,F1)}$ and $\tau_{(a,F2)}$, since during inheritance F2 is the actual for F1. For example assume that $\tau_{(a,F1)}$ is {Object, C_1} and $\tau_{(a,F2)}$ is {Object, C_2}. The common subtype of these two types, and thus the type parameter to SortedSet, is {Object, C_1, C_2}.

3.4 Notational Conventions

We use the following notation in the rest of the paper.

$\tau_{(\texttt{this},[\![a]\!])}$, $\tau_{(\texttt{this},X)}$, τ_1, τ_2, ρ, μ	Type variables
$\tau_{(a)}$	Abbreviation for $\tau_{(\texttt{this},[\![a]\!])}$
σ	Some Java type
Γ	Some type in Generic Java
T	Some Java class
X, K, V, E	Formal generic parameters
a, b, u, w, x, v, t	Variables

4. BASIC ALGORITHM

To illustrate our algorithm, we will use the class in Listing 6 as a running example. At a high level Ilwith works as follows:

1. Replace all types in the classes to be generified with type variables (Section 4.1).
2. Generate constraints by applying the constraint generation rules to the analyzed classes (Section 4.2)
3. Merge all type variables that can be merged without loss in precision (Section 4.3)
4. Select which type variables will become formal type parameters (Section 4.4)
5. Calculate bounds for the type parameters (Section 4.5)
6. Instantiate the other type variables (Section 4.6)

The above algorithm works on individual or groups of Java classes. For best results, a class should be analyzed with or after all classes that it depends on.

4.1 Replacing Java Types with Type Variables

This step replaces all Java types with type variables. Ilwith uses the naming scheme in Section 3.2 to come up with the type variables. Listing 7 shows our running example after replacing Java types with type variables.

4.2 Generating Constraints

Ilwith generates constraints by processing the code of the analyzed classes. Since Ilwith does not require a whole-program analysis, some of the references it encounters during constraint generation may refer to classes other than the ones being analyzed (*foreign* classes). For this reason, Ilwith uses two constraint generation strategies: one that involves only classes in the set of classes currently being analyzed and the other that involves a foreign class.

Recall that our representation guarantees that each statement has at most one dereference (i.e., access to a field). We do not consider accesses to local variables and instance variables of "this" as a dereference. We use the following decision procedure for determining which set of constraint generation rules to use for a given statement:

- If the statement explicitly refers to a Java type (e.g., assigns a string literal or is an invocation of "new") use the rules in Section 4.2.1.
- If the type of the variable dereferenced is in the set of currently analyzed classes, use the rules described in Section 4.2.2. If there is no dereference expression in the statement (i.e., only local variables and instance variables of "this" are accessed), also apply these rules after inserting dereferences of this as necessary to match the rules. As $\tau_{(this,x)}$ is the same as $\tau_{(x)}$ we are free to use whichever version is more convenient for a constraint generation rule if more than one is applicable.
- If the type of the variable dereferenced is a foreign class, use the rules in Section 4.2.3

Listing 6: Running Example

```
class S {
  S next;
  Comparable val;
  List val2;

  void setVal(Comparable to) {
    val = to;
    next.val = to;
    val2.add(to);
  }
  Comparable getVal() {
    return val;
  }
};
```

Listing 7: Running example after introducing type variables

```
class S {
  τ(next) next;
  τ(val) val;
  τ(val2) val2;

  void setVal(τ(to) to) {
    val = to;
    next.val = to;
    val2.add(to);
  }
 τ(getValRet) getVal() {
    return val;
  }
};
```

$$\frac{a = \textit{new}\ \texttt{T}()}{\texttt{T} <: \tau_{(a)}}$$

$$\frac{a = \text{"}\langle\texttt{String}\rangle\text{"}}{\texttt{String} <: \tau_{(a)}}$$

Definition 1: Constraints due to explicit type references

Listing 8: A class benefiting from distinguishing the variable used to access it

```
class Storage {
  Object data;
  Storage a, b;

  void do_something() {
    a.data = new A();
    b.data = new B();
  }
}
```

4.2.1 Constraints for Explicit Type References

Definition 1 gives a sampling of the rules for generating constraints that arise from explicit type references, such as allocation and assignment of literals. For example, after `a = new T()`, we place the constraint that `T` $<: \tau_{(a)}$ (i.e., `T` is a subtype of $\tau_{(a)}$). Similarly, after `a = "hello"`, we place the constraint that `String` $<: \tau_{(a)}$.

4.2.2 Constraints for Accesses to Analyzed Classes

Definition 2 gives the constraint generation rules for accesses involving only analyzed classes. Note that our type-name notation qualifies accesses to fields with the variable used to get to the field (e.g., $\tau_{(x,v)}$). To see the value of this, consider the example in Listing 8: if we did not distinguish between the `data` fields of `a` and `b` using type names $\tau_{(a,data)}$ and $\tau_{(b,data)}$ we would end up using the same type for the `data` fields of all `Storage` objects, which is clearly imprecise.

4.2.3 Constraints for Accesses to Foreign Classes

If the type of the variable dereferenced is not in the set of classes being analyzed, and thus of a foreign class, there are three possibilities:

1. The type of the dereference is a normal Java type (e.g., `String`)
2. The type of the dereference is a formal generic parameter of the foreign class.
3. The type of the dereference is an instantiation of a generic class.

Definition 3 gives the rules for generating the constraints for the above cases. This definition uses a helper function Ψ which has three cases, one for each of the possibilities above. We now describe the three cases in detail.

Constraints for "Normal" Java Types
The first case of Definition 4 gives the constraint generation rules when the foreign access evaluates to a normal Java type.

$$\frac{a = x.v \quad x.v : \tau_{(v)}}{\tau_{(x,v)} <: \tau_{(a)}}$$

$$\frac{x.v = a \quad x.v : \tau_{(v)}}{\tau_{(a)} <: \tau_{(x,v)}}$$

$$\frac{a = x.f(\ldots) \quad x.f : \tau_{(p_1)}, \ldots, \tau_{(p_m)} \rightarrow \tau_{(f_{ret})}}{\tau_{(x,f_{ret})} <: \tau_{(a)}}$$

$$\frac{x.f(a_1, \ldots, a_m) \quad x.f : \tau_{(p_1)}, \ldots, \tau_{(p_m)} \rightarrow \tau_{(f_{ret})}}{\forall n._{1\le n\le m} : \tau_{(a_n)} <: \tau_{(x,p_n)}}$$

Definition 2: Constraints for accesses of analyzed classes

$$\frac{a = u.t \quad u.t : \Gamma}{\Psi(a, u, \Gamma, :>)}$$

$$\frac{u.t = a \quad u.t : \Gamma}{\Psi(a, u, \Gamma, <:)}$$

$$\frac{a = u.f(\ldots) \quad u.f : \Gamma_1, \ldots, \Gamma_m \rightarrow \Gamma}{\Psi(a, u, \Gamma, :>)}$$

$$\frac{u.f(a_1, \ldots, a_n) \quad u.f : \Gamma_1, \ldots, \Gamma_m \rightarrow \Gamma}{\forall n._{1\le n\le m} : \Psi(a_n, u, \Gamma_n, <:)}$$

Definition 3: Constraints for dereferenced accesses of foreign classes

For an example consider `a = u.t`, where `a` is a variable in the analyzed class and `t` is a `String`-typed field of a foreign class. In this case we generate the constraint `String` $<: \tau_{(a)}$.

Constraints for Formal Generic Parameters
The second case of Definition 4 gives the constraint generation rules when the type of the dereference is a formal generic parameter of the foreign class.

To see the motivation for this case, imagine that the analyzed class uses a variable `u` declared to be of type `Stack` which has a formal type parameter called `E` and a method `E top()`. Assume now the assignment `a = u.top()` where `a` is a variable in an analyzed class and `u` declared to be of type `Stack`. In this case, we would like to pose the constraint `E` $<: \tau_{(a)}$. However, this is too imprecise: Consider two unrelated stack variables, `u` and `w`. After the assignments `a = u.top()` and `b = w.top()`, we will end up generating the constraints `E` $<: \tau_{(a)}$ and `E` $<: \tau_{(b)}$. In other words, we are constraining the types of `a` and `b` by the same type of `E` even though there is no relationship between `a` and `b`.

What we really want to do is to constrain `a` and `b` by the actual type that is passed to `E` in the declarations of `u` and `w` (e.g., `Stack⟨Integer⟩ w; Stack⟨Boolean⟩ u;`). To do that, we use a type variable of the form $\tau_{(x,E)}$ (Section 3.3). For example, $\tau_{(u,\texttt{E})}$, and $\tau_{(w,\texttt{E})}$ represent the actual types passed to the formal `E` on instantiations for `u` and `w` respectively.

$$\Psi(a,u,\Gamma,R) := \begin{cases} (1) \quad \tau_{(\texttt{a})}\ R\ \sigma & \Gamma \text{ is normal type } \sigma \\ (2) \quad \tau_{(\texttt{a})}\ R\ \tau_{(u,X)} & \Gamma \text{ is formal generic parameter X} \\ (3) \quad \tau_{(\texttt{a})}\ R\ \sigma, & \Gamma \text{ is parameterized type } \sigma\langle\ldots\rangle \\ \quad\quad \forall (Y,\Gamma') \in \mathit{FormalBindings}(\Gamma): & \text{with } \Gamma' \text{ actual of formal generic parameter Y} \\ \quad\quad\quad \tau_{(\texttt{a},Y)}\ R\ \sigma' & \quad \Gamma' \text{ is normal type } \sigma' \\ \quad\quad\quad \tau_{(\texttt{a},Y)}\ R\ \tau_{(u,X')} & \quad \Gamma' \text{ is formal generic parameter X'} \end{cases}$$

Definition 4: Definition of Ψ

In the example above (`a = u.top()`) we would generate the constraint $\tau_{(u,\texttt{E})} <: \tau_{(a)}$.

Constraints for Parameterized Classes
The third case of Definition 4 gives the constraint generation rules when the type of the dereference is an instantiation of a class.

To see why we need this rule, consider the assignment `a = u.iterator()`, with `iterator()` declared to have the return type `Iterator⟨E⟩`, where `E` is a formal generic parameter of the class of `u`. Moreover, let's suppose that `u.iterator()` returns `Iterator⟨Integer⟩`. If we did not have the third case, we would generate only the constraint `Iterator⟨...⟩` $<: \tau_{(a)}$. While this constraint is correct, it does not capture the relationship between the type parameter passed to `Iterator` on `a`'s declaration and `Integer` when determining the type of `a` (Section 4.6). The third case in Definition 4 explicitly generates the constraint `Integer` $<: \tau_{(a,E)}$ to capture that relationship.

For ease of explanation the definition of Ψ in Definition 4 is slightly simplified. Specifically, it does not work when a generic class is instantiated with an instantiation of another generic class, such as `Set⟨Iterator⟨Integer⟩⟩`. Our system uses a slight variation of Ψ which uses recursion to handle such cases.

4.2.4 Constraints for the Running Example

Ilwith generates the following constraints using the rules in Definition 2: (i) $\tau_{(\texttt{to})} <: \tau_{(\texttt{val})}$ for the statement `val = to`; (ii) $\tau_{(\texttt{to})} <: \tau_{(\texttt{next,val})}$ for the statement `next.val = to`; and (iii) $\tau_{(\texttt{val})} <: \tau_{(\texttt{getValRet})}$, for the statement `return val`.

To pick the correct constraint generation rule from Definition 4 for `val2.add(to)`, we need to examine the declaration of `val2`. Assuming that the `List` used is declared as `List⟨E⟩` with the method `void add(E)`, we will chose rule 2 from Ψ, as the Γ in our case will be `X`. Therefore, we will generate constraint (iv) $\tau_{(\texttt{to})} <: \tau_{(\texttt{val2,E})}$.

4.3 Merging Type Variables

After generating the constraints, Ilwith analyzes the constraints and merges type variables that are in a strongly-connected component of constraints. For example, if we have the constraints $\tau_1 <: \tau_2, \tau_2 <: \tau_1$, we merge τ_1 and τ_2.

If Ilwith merges two variables $\tau_{(\texttt{this},[\![a]\!])}$ and $\tau_{(\texttt{this},[\![b]\!])}$ Ilwith considers a and b to be equivalent, that is $[\![a]\!] \equiv [\![b]\!]$. Once Ilwith merges a and b, it makes sense to also merge the type variables that refer to $[\![a]\!]$ with the type variables that refer to $[\![b]\!]$. More specifically, merging $[\![a]\!]$ and $[\![b]\!]$ causes $\tau_{(x,[\![a]\!])}$ and $\tau_{(x,[\![b]\!])}$ to be merged.

While the above merging strategy does not result in any loss in precision, sometimes it may make sense to merge even when there is a loss in precision. Section 6.2 presents optional merging strategies that lose precision but may actually enable Ilwith to produce better overall results.

Our running example does not have any opportunities for merging.

4.4 Selecting Type Parameters

Parameters to Java generic classes may be bound from above but not from below. For example, the following could be a legal declaration for a generic Java class:

```
class SortedList⟨Elt <: Comparable⟩
```

where `Elt` is the name of the type parameter to `SortedList` and `Comparable` bounds it from above (i.e., any type passed to `Elt` must be a subtype of `Comparable`). Bounding a type from below (e.g., requiring `Elt` to be a supertype of `Comparable`) does not make much sense and thus Java will not support it.

Thus, all type variables that are bound only from above and do not involve primitive types (which cannot be type parameters in Java) are candidate for type parameters. Using the basic algorithm, Ilwith converts all such type variables into type parameters. For our running example, there is only one type variable that is bound only from above: $\tau_{(\texttt{to})}$. Thus we select $\tau_{(\texttt{to})}$ to be a type parameter.

Section 6.1 extends this strategy by including optional heuristics based on "good" programming style.

4.5 Calculating the Bounds

To determine the bound on a type parameter, Ilwith takes the greatest common subtype of (i) all concrete (i.e., not type variable) types that constrain the type parameter; and (ii) the declared types of all variables represented by the type parameter. This greatest-common subtype becomes the constraint on the type parameter. For example, if the constraints on τ_1 are $\tau_1 <:$ `String` and $\tau_1 <: \tau_2$, and the declared type of the variable represented by τ_1 is `Object`, Ilwith determines the bound for the type parameter τ_1 by picking the common subtype of `String` and `Object`. Once Ilwith computes the bound, it can fix the remaining type variables (e.g., τ_2) based on the calculated bounds.

A formal generic parameter generated by this method can be used as a formal generic parameter with its calculated bound: As there were no subtype constraints on the original type variable, it is legal to substitute the formal with any subtype of the bound or the bound itself during instantiation of the new generic class. If there had been a statement that could be invalidated by this – i.e. assigning a concrete value to a location new represented by the formal – a subtype constraint would have been imposed on the original type variable.

To understand the validity of the bound, let's consider when it would be invalid. To be invalid, the new formal generic parameter would need to represent a location that was originally declared with a subtype of the newly calculated bound. However, as the bound includes the common subtype of all declarations represented by the new formal, this clearly cannot be the case.

For our running example, we need to calculate the bound for the one type parameter $\tau_{(\texttt{to})}$. As there are no explicit constraints on $\tau_{(\texttt{to})}$, we simply take the type of the single location it represents – `Comparable` – as a bound.

4.6 Determining Instantiations for Type Variables

As discussed earlier, we generate type variables (such as $\tau_{(z,a)}$) to represent the type of various locations in the program, including the actual type parameters to pass to generic classes at their instantiation. Some of these type variables are chosen by Ilwith to be formal generic parameters. However, for the remaining type variables, we need to calculate new types in order to generate a class that will typecheck. We also need to calculate parameters to pass to allocations of generic classes. For example, given `List x = new List()`, where `List` is a generic class with a type formal `E`, we need to determine the type parameters to pass to the declaration of `x` and to the allocation of the `List`. In this case, both type parameters are represented by the type variable $\tau_{(x,E)}$. The type variables considered in this section may be constrained from both above and below.

There are three steps to solving the constraints for a type variable: (i) find the greatest common subtype of all the supertype constraints (call it `T`); (ii) find the least common supertype of all the subtype constraints (call it `U`); (iii) any type on the path from `U` to `T` is a legal type to use. To lose as little information as possible, we pick `U` provided that it is a subtype of `T`. Choosing the most specific type here is a heuristic based on our intuition. One could, for example, choose `T` instead of `U` or alternately, Ilwith could try out the various possibilities for instantiations in an attempt to get at the "best" overall result. It is worth noting that due to type-unsafe code (Section 7) it may happen that Ilwith is unable to find any type with which to instantiate a type variable. In this case, Ilwith asks the user of the system for feedback. An alternate would be to use raw types [2] as used by Donovan *et al.* [5].

We now argue that our method for instantiating type variables is valid. First, the instantiation for a type variable will always be a subtype of the declared types of all locations represented by the type variable. Thus, all uses of the represented locations will remain valid. Second, the instantiation for a type variable will always be a supertype of all subtype constraints (which are caused by assignments to locations represented by the type variable). Thus, an assignment to the locations represented by the type variable will remain valid. Thus, Ilwith's instantiation of type variables will be valid in all cases.

For our running example, we need to calculate the types for the remaining type variables: Most of them ($\tau_{(\texttt{val})}$, $\tau_{(\texttt{getValRet})}$, $\tau_{(\texttt{next,val})}$, $\tau_{(\texttt{val2,E})}$) are constrained to be a supertype of $\tau_{(\texttt{to})}$, and thus of `Comparable`, and subtype of `Comparable` due to the declarations. Therefore we can choose `Comparable` or $\tau_{(\texttt{to})}$ for each of them. However since earlier Ilwith had already decided to not make these variables into type formals, we instantiate them with `Comparable` instead of $\tau_{(\texttt{to})}$.

We now need to instantiate $\tau_{(\texttt{val2})}$ whose only constraints come from its declared type (`List`$\langle\ldots\rangle$). Since the declared type is generic, we need to instantiate it. Thus, we look up

Listing 9: Running example after instantiating type variables

```
class S⟨X extends Comparable ⟩ {
  S⟨Comparable⟩ next;
  Comparable val;
  List⟨Comparable⟩ val2;

  void setVal(X to) {
    val = to;
    next.val = to;
    val2.add(to);
  }
  Comparable getVal() {
    return val;
  }
};
```

Listing 10: The internal per-method polymorphism

```
class Poly {
 static void poly( Object a, Object b ) {
  a = b;
  b = a;
 }

 void use() {
  Object x, y;

  poly(new A(),x);
  poly(new B(),y);
 }
}
```

the constraints on $\tau_{(\texttt{val2,E})}$ which represents the type parameter passed to `E` on the declaration of `val2`. Since $\tau_{(\texttt{val2,E})}$ is constrained only by its bound (`Comparable`), we instantiate the `List` in the declaration of `val2` with `Comparable`.

Finally, since `S` is now a generic type, we need to instantiate it in the declaration of `next`. Since $\tau_{(next,to)}$ is the formal type parameter to `S` in the declaration of `next` we look for constraints on $\tau_{(next,to)}$. The only constraint on $\tau_{(next,to)}$ comes from the bound on the formal generic parameter (i.e., `Comparable`) and thus we instantiate `S` with `Comparable` in the declaration of `next`.

The above instantiations give us the representation in Listing 9. While Listing 9 makes some use of generics, it is not what a programmer would have produced. The heuristics in Section 6 address the problems with this output.

4.7 Per-Method Polymorphism

Without per-method polymorphism, Ilwith will merge information from different call sites, resulting in a loss in precision. Ilwith supports optional per-method polymorphism for methods of analyzed and non-analyzed classes:

While imposing the constraints on methods belonging to analyzed classes, Ilwith generates new type variables representing the arguments and the return value of the method for each call-site of the method. If the original type variables are merged, Ilwith will merge the corresponding copies.

To see the benefit of this approach, consider the example shown in Listing 10. If one choses not to use per-method polymorphism, Ilwith imposes the constraints $\tau_{(x)} <: \tau_{(b)}$, $\tau_{(y)} <: \tau_{(b)}$, `A` $<: \tau_{(a)}$, and `B` $<: \tau_{(a)}$. Since $\tau_{(a)} <: \tau_{(b)}$

and $\tau_{(b)} <: \tau_{(a)}$ (from body of `poly`) $\tau_{(a)}$ and $\tau_{(b)}$ will be merged, producing the constraint system $\tau_{(x)} <: \tau_{(\llbracket a,b \rrbracket)}$, $\tau_{(y)} <: \tau_{(\llbracket a,b \rrbracket)}$, $\mathtt{A} <: \tau_{(\llbracket a,b \rrbracket)}$, $\mathtt{B} <: \tau_{(\llbracket a,b \rrbracket)}$. If one now applies a heuristic that causes these variables to be merged (see Section 6.2), `x` and `y` would end up being of a common subtype of `A` and `B`, a clearly unacceptable result.

However, if one uses per-method polymorphism the system will generate the constraints $\tau_{(x)} <: \tau_{(b_1)}$ and $\tau_{(y)} <: \tau_{(b_2)}$ as well as $\mathtt{A} <: \tau_{(a_1)}$ and $\mathtt{B} <: \tau_{(a_2)}$. As a merge of the original variables implies a merge of the corresponding copies, merging $\tau_{(a)}$ and $\tau_{(b)}$ will produce the constraint system $\tau_{(x)} <: \tau_{(\llbracket a_1,b_2 \rrbracket)}$, $\tau_{(y)} <: \tau_{(\llbracket a_2,b_2 \rrbracket)}$, $\mathtt{A} <: \tau_{(\llbracket a_1,b_1 \rrbracket)}$, $\mathtt{B} <: \tau_{(\llbracket a_2,b_2 \rrbracket)}$. As there now is no constraint that links $\tau_{(x)}$ and $\tau_{(y)}$ together, subsequent application of heuristics will not degrade the results.

If the polymorphic method resides outside the set of analyzed classes we can use a simpler approach. As we already know which parameters of the method have the same type parameter, we do not need to create copies of the actuals when calling a polymorphic method. Instead, we can use a special set of constraint generation rules (Definition 5) that directly use the generic signature rather than waiting for the representatives to be merged.

Note that the previous description has treated the feature as optional. Ilwith normally does not use per-method polymorphism, as long as it is not explicitly enabled on a per-method basis. Section 6.4 gives a heuristic that choses for which methods to use per-method polymorphism.

5. HANDLING FEATURES OF JAVA

We now describe how our algorithm handles some of the trickier aspects of analyzing Java programs.

5.1 Arrays

Ilwith models arrays as instances of the `java.lang.reflect.Array` class, which it treats as a generic class with one type parameter (`X extends Object`) representing its element type. Loads and stores into the array translate into invocations of the `set` and `get` methods with signature `void set(X, int)` and `X get(int)` respectively.

5.2 Public fields

Ilwith does not have access to all uses and modifications of public instance variables since any class may use or modify these variables. There are three possibilities for handling public instance variables: (i) Assume that a public field can only be of its declared type. This assumption may constrain the types of other variables in the class. Though sound, this alternative is too restrictive, since Java programmers, in our experience, frequently use public fields even in situations where such broad access is unnecessary; (ii) Assume that a public field will not be modified from outside the class. This is potentially unsound. (iii) Perform a whole program analysis (e.g., escape analysis) to figure out how public fields may be modified outside of their classes. Since whole program analysis of Java is in general impossible (due to dynamic class loading) this possibility has limited applicability. For our experiments we use the second strategy above.

5.3 Interfaces

Since interfaces do not contain any code, Ilwith cannot directly convert a non-generic interface into a generic interface. Ilwith handles interfaces by analyzing implementations of the interface. Each type variable of a implementation is declared to be equal to the type variable representing the same declaration in the interface. By doing this, the types in the interface will end up being types and formals that fulfill all requirements of all implementations. Ilwith uses a similar approach to convert abstract classes to be generic.

While the approach above works well in practice it is unsound (unless we can analyze all possible implementations of the interface).

5.4 Inheritance

If class `S` is a subclass of class `T`, Ilwith treats class `S` as if it had a (specially named) field of type `T`. Method invocations and variable accesses to ancestor classes are delegated to this field. Thus, Ilwith can instantiate superclasses in the same way that it instantiates types for instance and local variables.

5.5 Native Methods and Reflection

Since native methods are not written in Java, Ilwith cannot analyze them and therefore assumes the worst case about them. Since classes rarely call native methods we have not found the worst case assumption to be a problem in practice.

Ilwith handles some but not all uses of reflection. Consider for example the use of `Object[] newInstance(Class,int)` in `java.lang.relect.Array` to create a new instance. The problem here is that Ilwith needs to know the value of the first argument to `newInstance` to determine the return type of `newInstance`. To handle this situation, we assume that `Class` is a generic type and that the signature of `newInstance` method is `X[] newInstance(Class[X],int)`. In other words, `newInstance`'s return type is the same type with which `Class` is parameterized. The, however, does not explain how `Class` objects are constructed. Ilwith does this by providing special treatment for functions that generate these objects, such as `getClass`: When analyzing a invocation such as `a = x.getClass()`, Ilwith imposes the constraints $\mathtt{Class} <: \tau_{(a)}$ and $\tau_{(x)} <: \tau_{(a,X)}$. This ensures that the formal type of the Class object (the `X`) always has the same type as the value it is constructed from.

6. COARSENING HEURISTICS

As we saw in Section 4, our algorithm produces results that are too precise to be useful. In this section we present heuristics that Ilwith uses to coarsen its results. These heuristics are based on our study of programming style used in Eiffel's generic classes [9]. As such, these heuristics are not meant to be complete. Users of Ilwith can selectively enable or disable the heuristics or extend Ilwith with heuristics of their own.

Our current set of heuristics are based on the following observations about the style of the generic code that we examined:

> **Style observation 1:** Generic classes have a small number of type parameters and these type parameters are usually unconstrained or constrained with an abstract class or interface.

> **Style observation 2:** Generic classes use type parameters only if they give some benefit that cannot be easily obtained using subtype polymorphism.

$$\frac{b = f(\ldots, a, arg_n, \ldots) \quad f : \ldots, X, \sigma_n, \cdots \to X}{\tau_{(a)} = \tau_{(b)}}$$

$$\frac{f(\ldots, a, arg_n, \ldots, b, arg_m \ldots) \quad f : \ldots, X, \sigma_n, \ldots, X, \sigma_m, \cdots \to \sigma_{ret}}{\tau_{(a)} = \tau_{(b)}}$$

Definition 5: The constraints used for per-method polymorphism.

Style observation 3: Generic classes use their type parameters frequently throughout the class. Moreover, generic classes commonly have methods with the type parameter as argument or return type or have instance variables with the type parameter as its type.

Style observation 4: Type parameters are seldom used polymorphically inside a class. For example, if we have a variable whose type is determined by a type parameter, it is rarely assigned to a variable of a different type.

Style observation 5: Methods such as *equals* are usually used to compare objects of the exact same type.

6.1 Selection Heuristics

These heuristics determine when we use a type variable as a formal of the generic class. We have two such heuristics.

6.1.1 Selection Heuristic 1

Turn a type variable into a formal of the enclosing generic class if it fulfills at least one of the following requirements:

- It is used as a parameter type or the return type of at least two methods of the class.
- It is used as a parameter type and the return type for at least one method of the class.
- It is used as the type of at least one instance variable of the class.

The second criterion above is subsumed by the first criterion. However, we include both in our system to provide users more choices since the second criterion is more picky than the first.

This heuristic is motivated by style observation 3 (and to a lesser extent style observation 1).

6.1.2 Selection Heuristic 2

Select a type variable $\tau_{(a,X)}$ as a formal if it is a formal of another type variable γ that has already been selected as a formal generic parameter and neither $\tau_{(a,X)}$, nor any other variable $\tau_{(a,X_n)}$ representing a formal of γ is constrained. Furthermore, prevent γ from being used as a formal.

To motivate this heuristic, consider the class in Listing 11 and its desired generic version in Listing 12. Without the above heuristic Ilwith will not produce the output in Listing 12 because the class does not actually do anything with `data` or `arg`. Instead Ilwith will produce the output in Listing 13. Using the above heuristic allows Ilwith to produce the desired output.

Listing 11: Motivation for selection heuristic 2

```
class Client {
  void set( List arg ) {
    data = arg;
  }

   List get() {
    return data;
  }

   List data;
}
```

Listing 12: Desired output for Listing 11

```
class Client⟨Y <: Object⟩ {
  void set( List⟨Y⟩ arg ) {
    data = arg;
  }

   List⟨Y⟩ get() {
    return data;
  }

   List⟨Y⟩ data;
}
```

Listing 13: What Ilwith produces for Listing 11

```
class Client⟨X <: List⟨Object⟩⟩ {
  void set( X arg ) {
    data = arg;
  }

   X get() {
    return data;
  }

   X data;
}
```

$$\frac{a = x.\mathtt{clone}()}{\tau_{(a)} = \tau_{(x)}}$$

$$\frac{x.\mathtt{equals}(a)}{\tau_{(a)} = \tau_{(x)}}$$

Definition 6: The constraint generation rules for `equals` and `clone`

Listing 14: An example of the “equals” heuristic

```
class equalsProblem {
  boolean isSet( Object arg ) {
    return data.equals(arg);
  }

  void set( Object arg ) {
    data = arg;
  }

  Object get() {
    return data;
  }

  Object data;
}
```

6.2 Merging Heuristic

A merging heuristic merges type variables together so that the combined type variable can be a candidate for being made into a type parameter.

The heuristic is as follows: Merge two type variables τ_1 and τ_2, if $\tau_1 <: \tau_2$, and τ_1 as well as τ_2 are constrained to be subtypes of the same set of concrete types.

This heuristic is motivated by style observation 4. To see this, consider again the class of the simple example in Listing 2. Without the merging heuristic, Ilwith produces the class in Listing 3 which is probably not what a programmer would have written. The `set` method takes an argument of type τ and immediately widens it to `Object`, thus losing type information. The merging heuristic merges the type variables corresponding to the types of `data`, `get`, and `set` and results in the output in Listing 4.

6.3 Constraints for `equals` and `clone`

Methods such as `equals` in Java can take arguments of any type (as long as it is a class). Thus, one could use `equals` to compare objects of completely unrelated types. However, as stated in style observation 5, it is common for `equals` to be used only for the same type as the receiver. This heuristic forces the type of the argument and receiver to be the same. Definition 6 gives the constraint generation rules for `equals` and `clone`.

To see the effect of this heuristic consider the class in Listing 14. Without this heuristic, Ilwith will not discover any connection between the argument type of `isSet` and the argument type of `set`.

If a class uses `equals` to compare arguments of different types, Ilwith will still produce correct but possibly inferior results with this heuristic. One possibility is for the user to try both with and without the heuristic and pick the better result.

6.4 Polymorphic Functions

Java programs frequently use static functions. As these functions are shared throughout a system, it would be hard to generate a useful generic representation of a static method that could be used from all clients, using the many different actual parameterizations possible. Thus, we use per-method polymorphism for the static methods.

6.5 Running Example

We had the following constraints for our running example: (i) $\tau_{(\mathtt{to})} <: \tau_{(\mathtt{val})}$, (ii) $\tau_{(\mathtt{to})} <: \tau_{(\mathtt{next,val})}$, (iii) $\tau_{(\mathtt{val})} <: \tau_{(\mathtt{getValRet})}$ and (iv) $\tau_{(\mathtt{to})} <: \tau_{(\mathtt{val2,E})}$.

The merging heuristic merges $\tau_{(\mathtt{val})}$ and $\tau_{(\mathtt{getValRet})}$. As both the type variables being merged are of the form $\tau_{(\mathtt{this},a)}$, we need to declare $\llbracket\mathtt{val}\rrbracket \equiv \llbracket\mathtt{getValRet}\rrbracket$ (Section 4.2). As a result of the merge we get $\tau_{(\llbracket\mathtt{val,getValRet}\rrbracket)}$. It is important to note that, as a result of declaring $\llbracket\mathtt{val}\rrbracket \equiv \llbracket\mathtt{getValRet}\rrbracket$, constraint (ii) now becomes $\tau_{(\mathtt{to})} <: \tau_{(\mathtt{next},\llbracket\mathtt{val,getValRet}\rrbracket)}$.

After performing this merge, we now have the constraint system: (i) $\tau_{(this,\mathtt{to})} <: \tau_{(this,\llbracket\mathtt{val,getValRet}\rrbracket)}$, (ii) $\tau_{(this,\mathtt{to})} <: \tau_{(next,\llbracket\mathtt{val,getValRet}\rrbracket)}$, (iii) $\tau_{(\mathtt{to})} <: \tau_{(\mathtt{val2,E})}$.

The merging heuristic is applicable again this time to $\tau_{(this,\mathtt{to})}$ and $\tau_{(this,\llbracket\mathtt{val,getValRet}\rrbracket)}$, yielding $\tau_{(this,\llbracket\mathtt{val,getValRet,to}\rrbracket)}$. Having done this, we then can collapse the constraint $\tau_{(this,\llbracket\mathtt{val,getValRet,to}\rrbracket)} <: \tau_{(next,\llbracket\mathtt{val,getValRet,to}\rrbracket)}$, ending up with the type variable $\tau_{(\llbracket this,next\rrbracket,\llbracket\mathtt{val,getValRet,to}\rrbracket)}$. The sole remaining constraint now is $\tau_{(\llbracket this,next\rrbracket,\llbracket\mathtt{val,getValRet,to}\rrbracket)} <: \tau_{(\mathtt{val2,E})}$, which we can merge too. The remaining type variable $\tau_{(\llbracket this,next,val2\rrbracket,\llbracket\mathtt{val,getValRet,to,E}\rrbracket)}$ clearly fulfills the requirements of the selection heuristic, so we chose to make it a formal generic parameter.

We now need to calculate the bound for the type formal to the generic class. As we have no concrete constraints - other than the declarations of the locations it represents - we use the bound `Comparable`.

Finally we need to instantiate the remaining type variables. As previously, the type variable $\tau_{(\mathtt{val2})}$ is unconstrained, so the type chosen will be $\mathtt{List}\langle\ldots\rangle$ again. Similarly, we look for $\tau_{(\mathtt{val2,E})}$ to determine the type used to instantiate the list. Although we do not find the type variable itself, we find something that has been declared equivalent to it: $\tau_{(\llbracket this,next,val2\rrbracket,\llbracket\mathtt{val,getValRet,to,E}\rrbracket)}$. As this type variable had been made into a formal generic parameter, we instantiate the `List` with the newly generated formal parameter, `X`.

Finally we need to instantiate `S` in the declaration of `next`. As our formal generic parameter represents the locations `val`, `getValRet` and `to`, we need to locate the type variable $\tau_{(\llbracket this,next,val2\rrbracket,\llbracket\mathtt{val,getValRet,to,E}\rrbracket)}$ to calculate the instantiation. Incidentally, this is exactly the type variable we chose to make a formal generic parameter - allowing us to simply chose the formal itself, generating the final program shown in Listing 15.

7. RESULTS

Ilwith operates on the `jimple` representation of the Soot framework [18]. Jimple is a high-level intermediate representation for Java programs that already provides most of the simplifications discussed in Section 3.1. The inputs to Ilwith are one or more compiled Java classes and a description of classes that are already generic. The output of Ilwith is a new description, which is a superset of the input file that also includes information for making the input classes generic.

Listing 15: Running Example

```
class S⟨X extends Comparable ⟩ {
  S⟨X⟩ next;
  X val;
  List⟨X⟩ val2;

  void setVal(X to) {
    val = to;
    next.val = to;
    val2.add(to);
  }
  X getVal() {
    return val;
  }
};
```

Class		Signature
ArrayList	(Java 1.4)	⟨E extends Object⟩
Vector	(Java 1.4)	⟨E extends Object⟩
Stack	(Java 1.4)	⟨E extends Object⟩
HashSet	(Java 1.4)	⟨E extends Object⟩
TreeMap	(Java 1.4)	⟨K, V extends Object⟩
HashMap	(Java 1.4)	⟨K, V extends Object⟩
LinkedList	(Java 1.4)	⟨E extends Object⟩
Vector	(Antlr)	⟨E extends Object⟩
LList	(Antlr)	⟨E extends Object⟩

Table 1: Discovered type parameters

7.1 Converting Non-Generic Classes to Generic Classes

To evaluate Ilwith's ability in converting non-generic classes to generic classes, we applied it to several data structures taken from the Sun Java 1.4 and Antlr class hierarchies. For all our test classes, Ilwith was fast, taking less than two minutes on a 2.66 Ghz Pentium 4 workstation with 2 GB of memory. Table 1 gives the signatures that Ilwith discovers for classes in the "Class" column. All the discovered signatures are correct and are consistent with what we would have done manually. For all classes except for `TreeMap` and `HashMap`, Ilwith converted the classes fully automatically. We discuss the issues with `TreeMap` and `HashMap` in the remainder of this section.

For the `HashMap`, Ilwith initially reported the signature `HashMap`⟨$\tau_{(val)}$ `extends HashMap$Entry`⟨ `Object`, $\tau_{(val)}$ ⟩⟩, rather than the expected `HashMap`⟨$\tau_{(key)}$, $\tau_{(val)}$ `extends Object`⟩. To find the cause for this output, we instructed Ilwith to calculate the formals and instantiations for each type variable before and after each merge. Using this output, it is easy to locate the causes of $\tau_{(val)}$'s strange signature: One simply locates the first merge after which the calculated bound of $\tau_{(key)}$ differs from the desired one. As each merge is caused by a constraint, which, in turn was generated by some statement, one then can inspect the statement that caused the undesired merge. Applying this method, we found the statement in the `HashMap` that had imposed the constraint to be the `return e;` in the following fragment:

```
if (e == null)
      return e;
if (e.hash == hash && eq(k, e.key))
      return e.value;
```

The intended behavior of this code is as follows: if the entry `e` is `null`, it should return `null`; if the entry is non `null` and matches the key, `k`, it should return the value stored at the entry. Looking at the code we see that rather than having a `return null` when the `e` is `null`, it returns `e` itself. Thus, the code returns values of two different types: the type of the entry and the type of the value stored in the entry. This issue is reflected by the output generated by Ilwith , as choosing the "undesired" bound will not generate any typing errors. However, if we modify this code to return the constant `null` Ilwith produces the expected bound.

After resolving this issue, we found that our system still did not create a generic parameter for the key of the `HashMap`. On investigation we found that the `HashMap` uses an instance of the `Object` class (allocated in the `HashMap` code) as the "empty" key. This assignment forced the key's type to be a supertype of `Object` which disqualified it from being a parameter. After removing the second issue from the class, Ilwith did discover the desired generic representation of the `HashMap` and the dependent classes.

Besides issues similar to the ones for `HashMap`, we found additional problems with `TreeMap`. Ilwith initially reported a generic signature of `TreeMap$Entry`⟨$\tau_{(data)}$`extends TreeMap$Entry`⟨...⟩⟩ for the `TreeMap$Entry` class. This is clearly problematic since it would allow us to store only objects of type `TreeMap$Entry` in the `TreeMap`

We found the reason for this signature to be as follows. Internally, the `TreeMap` stores key/value pairs inside the helper class `TreeMap$Entry`. `TreeMap` also defines a class `TreeMap$EntryIterator`, for iterating over all entries stored in the tree map. This iterator returns `TreeMap$Entry` from its `next` method. The `TreeMap` also defines two subtypes of the `EntryIterator`, one for iterating over the values stored in the map and one for iterating over the keys in the map. Similarly, these iterators return values and keys from their respective `next` method. Even though the `next` methods of the three iterators return completely unrelated types, the inheritance and overriding forces them all to return the same type. It is worth noting that, although the `HashMap` uses inheritance similarly, it it does not exhibit the above problem, since the code for `HashMap` does not mix unrelated types.

Even after we had addressed the above issue in the code of `TreeMap`, Ilwith was still unable to infer the desired type parameters for `TreeMap`; it inferred the keys of the map to be of type `TreeMap$Entry`. The reason for this turned out to be a method, `buildFromSorted`, in `TreeMap`. `TreeMap` uses this method as an efficient way to construct a new tree if sorted data is already available, e.g. in the case of a clone. `buildFromSorted` takes an iterator that returns the key/-value pairs in the correct order for constructing the tree. However, `buildFromSorted` also contains special support for the deserialization of the `TreeSet`: it enables the iterator to return both keys and key/value pairs. Thus, Ilwith concludes that the iterator may return either `TreeMap$Entry` or `Object` (from the key) and thus infers the incorrect type for the key. When we commented out the special support for `TreeSet` Ilwith found the desired generic signature for `TreeMap` and its dependent classes.

To summarize, Ilwith is effective in converting classes to use generics. When it failed it was due to weak typing in the original Java classes. No matter how these classes are converted to be generic, whether using a tool or manually, the issues above will have to be fixed.

Disabled	None	Merging	Selection	`equals`	All
`ArrayList`	1	6	8	1	24
`LinkedList`	1	5	10	1	48
`HashMap`	2	6	19	4	99

Table 3: Effectiveness of the heuristics wrt. the number of discovered type parameters

Disabled	None	Merging	Selection	`equals`	All
`ArrayList`	15	6	16	12	20
`LinkedList`	32	16	37	28	42
`HashMap`	2	6	19	4	99

Table 4: Effectiveness of the heuristics wrt. the number of times a type parameter is used in the class

7.2 Instantiating Generic Classes

Table 2 shows sample instantiations that Ilwith computed in our benchmark classes. In addition to fields and methods, entries with `extends` denote instantiations that were computed for superclasses. ⟨`init`⟩ denotes a signature of a constructor. All instantiations were correct.

7.3 Effectiveness of Heuristics

Table 3 presents the number of type parameters of the discovered generic class with different heuristics. Each column presents the data when one or more of the heuristics is *disabled*.[1] At two extremes, "None" gives the results of the default configuration, where no heuristic is disabled, and "All" gives the results where all of the heuristics are disabled.

Table 4 gives the number of functions and fields that get their type from a type parameter. From the "None" column we see that even though the classes have one or two type parameters, they are used in many places.

From these tables we see that all heuristics offer some benefit. "equals" offers the least benefit while the merging and selection heuristics are most beneficial. However, to achieve the desired result our benchmarks, we had to enable all the heuristics.

7.4 Complexity of the Problem

Figures 1 and 2 show a more detailed view of the LinkedList and HashMap test cases. Each node represents an analyzed class while an edge goes from a class to classes used by that class.

From these figures we see that even seemingly simple classes, such as `LinkedList`, are complicated and use many inner classes. In the process of analyzing the classes in Table 1 we analyzed a total of 47 classes (which includes inner classes and abstract classes). We also see that there are complex dependencies between classes and there are several strongly-connected components (indicating circular dependency) in the two graphs. These circular dependencies validate our decision to combine the tasks of instantiating classes and converting non-generic classes to generic classes.

8. RELATED WORK

We are aware of only of three pieces of prior work on converting non-generic Java classes to generic classes. However,

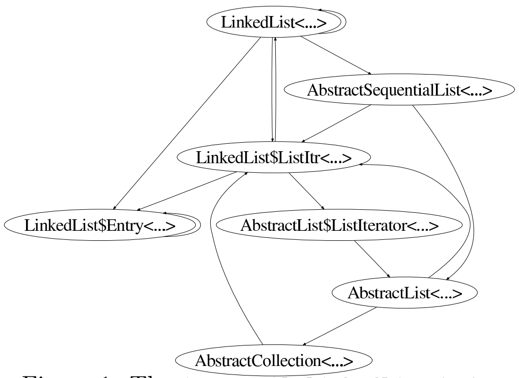

Figure 1: The `java.util.LinkedList`-test.

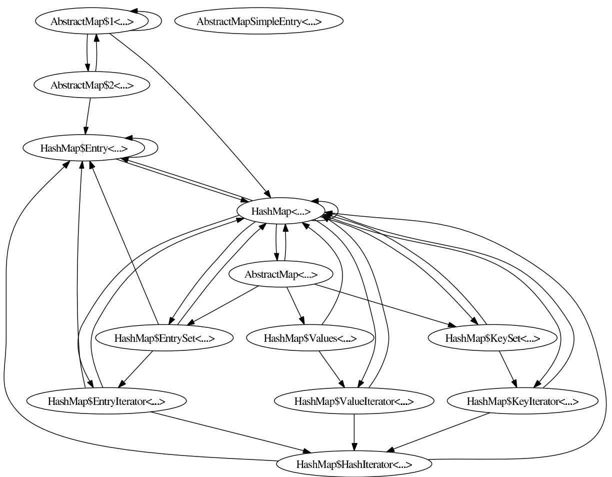

Figure 2: The `java.util.HashMap`-test.

much of the work on type inference, constraint-based analyses, and refactoring is also relevant to our paper.

Duggan [6] describes an analysis for converting Java classes to use generics. Like our work, Duggan's approach is based on type inference. However, unlike us, their analysis does not help in the instantiation of already generic classes. Also, unlike us they do not present any experimental results or recognize the need for incorporating heuristics into their analysis in order to yield useful results.

Donovan *et al.* [5], in concurrent work, describe a system for instantiating already generic classes. Unlike our analysis, their system does not convert non-generic classes to generic classes. The analyses used by Donovan *et al.* are more aggressive than ours, requiring whole-program points-to analysis. Donovan *et al.*'s analysis essentially does the Cartesian Product whole program analysis [1] to determine the space of possible solutions and then a type inference that picks between the possible solutions. Since whole program analysis is not, in general, feasible for Java programs (with dynamic class loading) our analysis is more generally applicable. On the other hand, Donovan et al. also generate uses of raw types [2] which our analysis does not consider. It would be worthwhile to try to combine the strengths of Donovan *et al.*'s analysis with ours.

Tip *et al.* [16], also in concurrent work, describes a system for instantiating generic container classes. Unlike us, Tip *et al.* do not try to convert non-generic classes to generic

[1]Due to implementation artifacts we could not disable the *polymorphic functions* heuristic completely.

Signatures calculated in **LinkedList⟨E extends Object⟩**	
	`extends` AbstractSequentialList⟨E⟩
E	set(int,E)
boolean	addAll(Collection⟨E⟩)
Array⟨E⟩	toArray(Array⟨E⟩)
LinkedList$ListItr⟨E⟩	listIterator()
LinkedList$Entry⟨E⟩	addBefore(E,LinkedList$Entry⟨E⟩
LinkedList$Entry⟨E⟩	header
Signatures calculated in **LinkedList$Entry⟨E extends Object⟩**	
	⟨init⟩(E,LinkedList$Entry⟨E⟩,LinkedList$Entry⟨E⟩)
element	E
LinkedList$Entry⟨E⟩	next
Signatures calculated in **LinkedList$Iterator⟨E extends Object⟩**	
	`extends` AbstractList$ListIterator⟨E⟩
	⟨init⟩(LinkedList⟨E⟩)
E	next()
void	set(E)
Signatures calculated in **HashMap$EntrySet⟨K,V extends Object⟩**	
	⟨init⟩(HashMap⟨K,V⟩)
void	remove(HashMap$Entry⟨K,V⟩)
HashMap$EntryIterator⟨K,V⟩	iterator()
Signatures calculated in **HashMap$HashIterator⟨K,V extends Object⟩**	
	⟨init⟩(HashMap⟨K,V⟩)
HashMap$Entry⟨K,V⟩	current
Signatures calculated in **HashMap$KeyIterator⟨K,V extends Object⟩**	
	`extends` HashIterator⟨K,V⟩
	⟨init⟩(HashMap⟨K,V⟩)
K	next()
Signatures calculated in **HashMap$KeySet⟨K,V extends Object⟩**	
	⟨init⟩(HashMap⟨K,V⟩)
boolean	contains(K)
boolean	remove(K)
HashMap$KeyIterator⟨K,V⟩	iterator()
Signatures calculated in **HashMap$Entry⟨K,V extends Object⟩**	
V	getValue()
K	getKey()
void	recordRemoval(HashMap⟨K,V⟩)
boolean	equals(HashMap$Entry⟨K,V⟩)
Signatures calculated in **HashMap⟨K,V extends Object⟩**	
	`extends` AbstractMap⟨K,V⟩
void	putAll(Map⟨K,V⟩)
V	get(K)
V	put(K,V)
void	createEntry(int,K,V,int)
HashMap$Entry⟨K,V⟩	removeEntryForKey(K)
void	transfer(Array⟨Entry⟨K,V⟩⟩)
Array⟨Entry⟨K,V⟩⟩	table
HashMap$KeySet⟨K,V⟩	keySet()
HashMap$EntrySet⟨K,V⟩	entrySet()
HashMap$KeyIterator⟨K,V⟩	newKeyIterator()
HashMap$EntryIterator⟨K,V⟩	entryIterator()
HashMap$ValueIterator⟨K,V⟩	newValueIterator()
Signatures calculated in **TreeMap⟨K,V extends Object⟩**	
void	buildFromSorted(int, Iterator⟨Map$Entry⟨K,V⟩⟩, IOStream, V)
V	put(K,V)
TreeMap$Entry⟨K,V⟩	parentOf(TreeMap$Entry⟨K,V⟩)
Signatures calculated in **LList⟨E extends Object⟩**	
E	deleteHead()
LLEnumeration⟨E⟩	elements

Table 2: Some discovered signatures

classes. Tip *et al.*'s approach is similar to Donovan *et al.*'s approach in that it starts with the Cartesian Product algorithm. Tip *et al.*'s approach uses the results of the Cartesian Product algorithm to identify the calling contexts and does a type inference that uses the calling context information.

The work on type inference and constraint based analyses [7, 15, 14] all influenced our work. To some degree our approach can be considered an application of constraint based techniques from prior work. Lackwit [14] uses a constraint-based analysis to detect possible errors in C programs. Lackwit identifies variables, based on assignments in the program, that must have an identical type and representation. By examining these types the programmer can detect erroneous flows of values. The output generated by Ilwith can also be used in a manner similar to Lackwit, as seen in the experimental results. Type inference in Standard ML [11] also aims at discovering parametric polymorphism in SML programs. We found that constrained-based type inference alone was not enough for converting classes to use generics: we also needed to use heuristics to yield desirable results.

Tip, Kiezun and Bäumer describe an approach to refactoring Java programs [17]. The refactorings they describe including extracting interfaces and moving class members. The approach used for the refactoring uses type constraints that are used in a similar fashion to our approach. The constraint generation and resolution used in our system can be viewed as a special case of their approach. However, we demonstrate that constraint generation and resolution alone is inadequate: one also needs to incorporate heuristics in order to obtain generic classes that are similar to what programmers would manually produce.

9. CONCLUSIONS

The next major release of Java will include support for generic classes. Since generics have software engineering and performance benefits, programmers may wish to convert their legacy Java code to use generics. To aid in this task, we describe a system that automatically converts Java classes to use generics. This conversion has two parts: (i) determining the type parameters and their bounds to use for a generic version of a non-generic class; and (ii) rewriting clients of the newly-generic class to pass appropriate type parameters to the generic class. Our approach handles both of these tasks using the same basic mechanism, which is based on type inference. Our system analyzes one strongly connected component of the class-dependency graph at a time. Thus, it can be used to convert a non-generic class to a generic class without knowing anything about clients of the class. We show that our system is effective in that it usually produces the same output that an expert programmer would produce.

10. REFERENCES

[1] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 2–26. Springer-Verlag, 1995.

[2] G. Bracha, M. Odersky, D. Stoutamire, and P. W. adler. Making the future safe for the past: Adding genericity to the Jav a programming language. In C. Chambers, editor, *ACM Symposium on Object Oriented Programming: Syst ems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[3] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, December 1985.

[4] A. Diwan, K. McKinley, and E. Moss. Using types to analyze and optimize object-oriented programs. *ACM Transactions on Programming Languages and Systems*, 23(1):30–72, February 2001.

[5] A. Donovan, A. Kiezun, and M. D. Ernst. Converting java programs to use generic libraries. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, Vancouver, Canada, October 2004. ACM. To appear.

[6] D. Duggan. Modular type-based reverse engineering of parameterized types in Java code. In *Proceedings of the 14th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 97–113. ACM Press, 1999.

[7] M. Fähndrich and A. Aiken. Program analysis using mixed term and set constraints. In *Static Analysis Symposium*, 1997.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition.* Addison-Wesley, Boston, Mass., 2000.

[9] B. Meyer. Eiffel: a language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246, 1988.

[10] B. Meyer. *Eiffel: the language.* Prentice-Hall, Inc., 1992.

[11] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[12] G. Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall, New Jersey, 1991.

[13] N. Nystrom, A. L. Hosking, D. Whitlock, Q. Cutts, and A. Diwan. Partial redundancy elimation for access path expressions. *Software-Practice and Experience*, 31(6):577–600, 2001.

[14] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference, 1997.

[15] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[16] F. Tip, R. Fuhrer, J. Dolby, and A. Kiezun. Refactoring techniques for migrating applications to generic java container classes. Technical Report RC23238(W0406-045), IBM Research Division, June 2004.

[17] F. Tip, A. Kiezun, and D. Bäumer. Refactoring for generalization using type constraints. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 13–26, Anaheim, CA, October 2003.

[18] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework, 1999.