SInfer: Inferring Information Flow Lattices for Checking Self-Stabilization

Yong hun Eom and Brian Demsky
University of California, Irvine
{yeom,bdemsky}@uci.edu

***Abstract*—Self-stabilizing programs are guaranteed to recover from state corruption caused by software bugs or other events and eventually reach the correct state. Many real-world applications including embedded controllers and multimedia applications can be designed to make key components self-stabilizing.**

Type systems and static analyses can automatically check whether a program is self stabilizing. The existing approach for checking self-stabilization requires developers to manually annotate code. We present an annotation inference algorithm that automatically derives an initial set of annotations and therefore lowers the effort to build self-stabilizing systems. Our experience with the algorithm indicates that it effectively inferred annotations for our benchmarks.

I. Introduction

Self-stabilizing programs are guaranteed to recover from state corruption to reach the correct state after a bounded number of steps [3]. Statically checking self-stabilization provides a new approach for improving the reliability of certain classes of software systems. In general, software bugs or other events can corrupt a program's state. After the program's state is corrupted, key invariants may have been violated and the program can behave arbitrarily. Despite extensive testing of software in industry, unusual and not so unusual inputs commonly trigger bugs in the field.

Prior work presented a combination of a type system and static analyses that together could check if a program self-stabilizes with respect to rarely triggered software bugs and certain classes of transient hardware errors [8].

Designing programs to be self-stabilizing is intended to complement and not replace existing approaches to software reliability. For example, we expect that software developers would still perform the same testing procedures on programs. Self-stabilization then gives developers more mileage out of the same testing process — even if the testing process misses a bug, the effects of that bug are guaranteed to be limited in time. Of course, if the system receives further fault-revealing inputs, the user will have to wait for it to self-stabilize again. Even statically verified programs can benefit from self-stabilization, transient faults could potentially introduce erroneous values into an execution of a verified program.

A. Overview of Previous Work

SJava [8] is a system for checking self-stabilization. The key components of SJava are as follows:

i) **Arrange State into a Lattice:** The approach uses annotations to define a location lattice. The location lattice contains edges that constrain how information flows between memory locations in the program. The location lattice is acyclic, prohibiting cyclic information flows. In SJava, every memory location (fields, arrays and variables) has a location type in additional to its Java type. The programmer uses location type annotations to map memory locations to lattice locations.

ii) **Constrain Information to only Flow Down this Lattice:** A type checker then checks that assignments only flow information down this lattice(hereafter referred to as the *flow-down rule*). The approach enforces the flow constraint on both the explicit flows caused by assignments and the implicit flows caused by conditional branches.

iii) **Eviction:** A separate static analysis then ensures that the program evicts values from a given location within a bounded time.

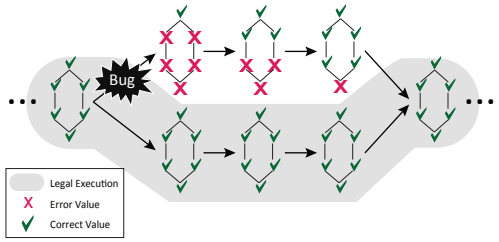

Fig. 1. Self-Stabilizing Execution Trace

Figure 1 graphically illustrates how these properties ensure self-stabilization. The red ×'s in the figure indicate corrupted values and the green ✓'s indicate correct values. The top trace shows the effect of a bug on a program execution while the bottom trace shows the corresponding desired bug-free execution. The effect of the bug is on the program's state is illustrated by the red ×'s in the figure. The bug then continues to affect the execution for a period of time as indicated by the divergence of the execution traces at the bug transition.

The self-stabilization checks ensure that after a period of time, the highest levels of the lattice must have correct values because they must be overwritten and can only depend on new input data. An inductive argument can then be made that if all levels above a given level have correct values, then after a bounded time period the given level must also have a correct value. The combination of the base case and the inductive case plus a location lattice of finite height ensures self-stabilization. This inductive argument can be seen graphically in Figure 1 as the red ×'s appear in lower portions of the location lattice until they finally disappear. Ultimately, this induction guarantees that the buggy execution converges with the correct execution.

B. Scope

Many programs maintain long term persistent state and are not inherently self-stabilizing. However, there are important domains that are likely to be self-stabilizing. These domains include embedded systems, multimedia codecs, and safety critical systems. Embedded systems often operate in environments that do not allow for frequent human intervention. Self-

stabilization can ensure that bugs never cause embedded controllers to transition into states in which they cannot function. Most multimedia codecs are at the design-level self-stabilizing as they must handle channel errors such as dropped frames caused by network failures or media damage. Self-stabilizing implementations of these codecs might fail to process short periods of a stream, but such failures will be transient and not affect the remainder of the stream.

C. Inferring Annotations

The previous work on checking self-stabilization required developers to manually annotate the program's code. While the number of annotations is not prohibitive, it does present a barrier to adoption. Moreover, annotating legacy systems requires first understanding how information flows through the program. Understanding legacy systems can be quite time consuming and present a significant barrier to adoption.

In this paper we present an approach for automatically generating annotations. Our approach is static — it infers annotations by analyzing the code. While the target of this paper is inferring information flow annotations for checking self-stabilization, the same type of information flow annotations can be used to check security properties.

Existing information flow inference systems [23], [24], [12], [11], [10] are not suitable for inferring SJava annotations. First, they only infer static types for objects while SJava allows an object to be created with one label and then transitioned to a different label. Second, because checking self-stabilization requires fine-grained types, many of the existing techniques would generate very complex annotations. Complex type annotations then complicate modification of the codebase as the developer is unsure of how to maintain the self-stabilization property of the code. Our goal is therefore to generate simple, human-understandable lattices. To avoid inferring overly complex annotations, we developed techniques that maintain precision where necessary while reducing the overall complexity of the lattices. Lastly, many existing techniques are insufficient for checking self-stabilization because they either track only explicit information flows [12], [11] or extract only the high-level security summaries [24], [10].

D. Contributions

This paper makes the following contributions:

- **New Approach to Building Robust Software:** This paper extends a new approach to building reliable software systems. It eliminates a key manual step in the process by automatically inferring the annotations for checking self-stabilization.
- **Annotation Inference Algorithm:** This paper presents an algorithm that can automatically infer a set of information flow annotations that are sufficient to check if a program self-stabilizes. Inferring information flow annotations is also potentially useful for checking security properties.
- **An Approach to Simplifying Annotations:** This paper presents a technique that simplifies the lattices. It improves the understandability of the lattices and may make our technique useful for other domains such as program understanding and debugging.
- **Implementation and Experience:** We have implemented our annotation inference algorithm and reported our experience with three benchmark applications.

II. Background on SJava

This section reviews the key ideas behind checking self-stabilization [8]. The two key ideas are: (1) to use a type system to arrange the program's state into a lattice of locations and then check that information only flows down this lattice and (2) bound how long information can remain at any given location in the lattice before it is evicted. The approach is targeted at programs that are structured with a main event processing loop. This is not a significant limitation as this is a common architecture for programs in the target domains. The developer annotates the event loop using the `SJAVA` loop label.

We use a weather index calculation example to illustrate the approach. Figure 2 presents the code for the example. The example program is structured as an event loop that reads the current temperature and humidity, computes an average temperature with the previous temperature, and then computes a weather index that combines temperature and humidity to determine the human-perceived temperature.

A. Location Types

The SJava location type system performs three functions. First, it provides a static location lattice that will be used to statically partition the program's state. Second, a set of type declaration annotations map the program's state onto locations in the location lattice. Together these components arrange the program's state and encode how information is allowed to flow between memory locations. Finally, the type rules check that all information flows in the program respect the location lattice. We next describe each of these components in more detail.

1) Location Lattice Definition: The location lattice defines the set of location types and orders them. SJava has two types of location lattices: *method lattices* and *field lattices*. Method lattices define a method local ordering of the methods' parameters and local variables.

Method lattices are declared with the annotation `@LATTICE`[1] that appears immediately before the corresponding method declaration. This annotation uses the lower than (<) operator to order two location types. Location types are implicitly declared by using them in an expression with the lower than operator in the lattice declaration. Line 11 of Figure 2 presents the method lattice for the `calculateIndex` method. The `@LATTICE` annotation declares two method location types, `IN` and `WEA`, and orders the type `IN` higher than the type `WEA`. This declaration means that information can only flow from locations with the type `IN` to locations with the type `WEA`. Each method has its own method lattice. This provides the necessary compositionality for libraries and other software components to be developed and annotated independently from the code that uses them. Compatibility of the method lattices between a caller and callee is then checked at the call sites.

Simply using the method lattice would result in all fields of a given object having the location of the object's reference. This would prevent any flow of information between different fields of the same object. To eliminate this limitation, SJava includes field lattices that order different fields of the same

[1] SJava does not add any new syntax to Java. Instead, SJava uses the existing Java annotation framework and loop labels for its annotations. As a result, SJava programs are legal Java programs.

```
1 @LATTICE("ATEMP<PTEMP,L1<ATEMP,L1<CHUM,L2<L1,
2  L3<L2,IDX<L3")
3 public class Weather {
4   @LOC("PTEMP") public float prevTemp;
5   @LOC("ATEMP") public float avgTemp;
6   @LOC("CHUM")  public float curHum;
7   @LOC("IDX")   public float index;
8   // define constants c1 to c9
9   public static final float c1=-0.22475541;
10  ...
11  @LATTICE("WEA<IN")
12  @THISLOC("WEA")
13  public void calculateIndex(){
14    SSJAVA:
15    while(true) {  // main event loop
16      @LOC("IN") float inTemp = Device.readTemp();
17      curHum = Device.readHumidity();
18      // calculate the average temperature
19      avgTemp = (prevTemp+inTemp)/2;
20      prevTemp = inTemp;
21      @LOC("WEA,L2") float f1=c1*avgTemp*curHum;
22      @LOC("WEA,L2") float f2=c2*avgTemp*avgTemp;
23      @LOC("WEA,L2") float f3=c3*curHum*curHum;
24      @LOC("WEA,L3") float f4=c4*f2*curHum;
25      @LOC("WEA,L3") float f5=c5*f3*avgTemp;
26      @LOC("WEA,L3") float f6=c6*f1*f2;;
27      index = c7+c8*avgTemp+c9*curHum+
28              f1+f2+f3+f4+f5+f6;
29 }}}
```

Fig. 2. Manually Annotated Weather Index Example

object. The field location lattice is declared using the same syntax as a method lattice, but the field lattice declaration appears immediately before a class declaration. Line 1 of Figure 2 declares the field lattice for the `Weather` class.

2) Type Declarations: SJava requires variable, parameter, and field declarations to have location types. The annotation `@LOC` is used to declare the location type of a parameter, field, or variable. The annotation `@LOC("IN")` in Line 16 of Figure 2 declares that the variable `inTemp` has the location type `IN`. The location type of a parameter is declared using the same syntax immediately before the type declaration of the parameter. The location type of the implicit `this` parameter is declared using the `@THISLOC` annotation. For example, the annotation `@THISLOC("WEA")` in Line 12 declares that the `this` variable has the location type `WEA`. The location types of fields are declared using the same syntax as variables. The annotation `@LOC("PTEMP")` in Line 4 of Figure 2 declares that the field `prevTemp` has the location type `PTEMP`.

3) Composite Locations: Method and field lattices are combined into a composite lattice with the ordering of the composite lattice determined by lexical ordering of the component lattices. Each type in a composite lattice begins with an element from the current method's location lattice followed by elements from field location lattices. For example, the location of a field with the field location type `IDX` in an object with the location type `WEA` is the composite location $\langle \texttt{WEA}, \texttt{IDX} \rangle$.

4) Checking Flows: Once the lattices are defined and the locations are declared, the flow-down check is implemented in the type checker. The SJava type checker checks that the right hand side of all assignments has a higher location than the left hand side. For array access expressions on the right hand side, the location is the greatest lower bound (GLB) of the array's location and the index expression. For arithmetic operations, the location is the GLB of the two operands. For array assignments, the location of the array must be lower than both the index and the right hand side. SJava handles implicit flows using a special program counter location. This location tracks implicit flows which result from conditional branches.

Each method has its own location lattice. The type checker must check at method calls that the parameters' location types are compatible with the arguments' location types.

B. Shared Locations

The flow-down rule is too restrictive for expressing many computations. Consider the following code excerpt:

```
1 for(int i=0;i<100;i++) ;
```

It is impossible to assign a location type to the variable `i` that will pass the flow-down rule check. SJava therefore includes *shared locations* to relax the flow-down constraint. Programs are allowed to have arbitrary flows between fields and variables with the same shared location. The only restriction is that there must exist a point in the execution of the event loop in which all locations with the same shared location have been overwritten with values from higher locations. Shared location types are denoted with a trailing * and only locations that store primitive values may have a shared location type.

C. Value Eviction and Other Checks

The flow-down rule by itself does not ensure self-stabilization. This rule may allow corrupt values to remain in a location indefinitely. To address this issue, SJava uses an eviction check, which ensures that locations are either (1) loop invariant, (2) overwritten in every loop iteration, or (3) overwritten before they are read. Satisfying any of these three properties ensures that the program cannot read stale, corrupt values. SJava uses static analysis to check these properties. This analysis does not require annotations and therefore is outside the scope of this paper.

Unrestricted aliasing could be used to circumvent the type system. SJava uses linear types[25] to ensure that a given object cannot be aliased by two different references with different heap locations. The linear type system analysis only requires a few delegate annotations and we expect that developers can easily write these without needing an inference tool. Currently, SJava prohibits recursive calls and recursive data structures[2].

III. Overview

SInfer automatically infer annotations for checking self-stabilization. We discuss both correctness properties and simplification goals for the inferred annotations.

A. Correctness Properties

We begin by stating the three correctness properties for SJava annotations:

i) **Lattice Structure**: The structure of the location types forms a lattice.

ii) **Completeness**: Every variable, field, method parameter, method return value, or method program counter must be assigned some type, whether implicitly or explicitly.

iii) **Flow Constraints**: All information flows are captured by the ordering relation constraints defined by lattices.

[2]Although the analysis design of the SJava supports recursive calls, the restriction is necessary because the termination analysis in SJava cannot currently check the termination of recursive calls.

B. Simplification Goals

Our initial implementation attempted to maintain maximally precise flow information that satisfied the correctness properties, but we found that the resulting lattices were extremely complex. This made the annotations incomprehensible to developers, and maintaining them during code revisions would not be feasible.

To reduce the complexity, we generally favor a lattice that defines the minimum number of location types necessary to maintain the correctness properties. However, this can unnecessarily constrain the external interfaces of classes and methods (e.g., parameters, return values, program counters and fields). The problem is that if the class or method were used in a different environment, developers may need to assign new location types. Therefore, for interface members, the location types should most accurately model the value flows in the original program. Thus, SInfer has the following two simplification goals:

i) **Precise Interfaces**: The location types for interface members must precisely model the value flows in the program.
ii) **Simplicity**: For all other locations, the lattice should be as simple as possible.

A key advantage of a simpler location type structure is that the developer can use the generated annotations to gain a basic intuition about how information flows through the program. With a simple structure, developers can easily figure out what location type constraints the code must satisfy to self-stabilize. Simplifying lattices is therefore useful for guiding the developer in modify the code in a way that maintains the self-stabilization property. In addition, if the developer chooses to customize the generated annotations, the simpler lattices will be easier to understand and modify.

IV. Annotation Inference Algorithm

In the previous section, we described three annotation correctness properties. To satisfy these properties, our basic approach attempts to maintain the most precise flow information. We first generate a set of constraints in the form of a directed graph where nodes are memory locations and edges represent explicit and implicit information flows. Similar to the constraint-based type inference [17], the graph captures constraints whose solution are the type annotations. Then, we perform inference and find mappings from locations to location types in the lattices that satisfy the set of constraints.

A. Value Flow Graph

The annotation inference algorithm begins by generating a *value flow graph* from the program source code. A node $n \in N$ in a value flow graph represents initial approximation of a location type assignment and is represented by a tuple $t = \langle v, \mathtt{f}_1, ..., \mathtt{f}_n \rangle \in T$ in which the first element of the tuple is a member of the method's variable lattice $v \in V$ and the subsequent elements are members of field lattices $\mathtt{f} \in F$. An edge $e \in E_f$ represents value flows.

Definition 1. *(Value Flow Graph) A method's value flow graph is a directed graph $G = (N, E_f)$, where a node corresponds to a location and an edge $(n_1 \rightarrow n_2) \in E_f$ corresponds to an explicit or implicit information flow from n_1 to n_2.*

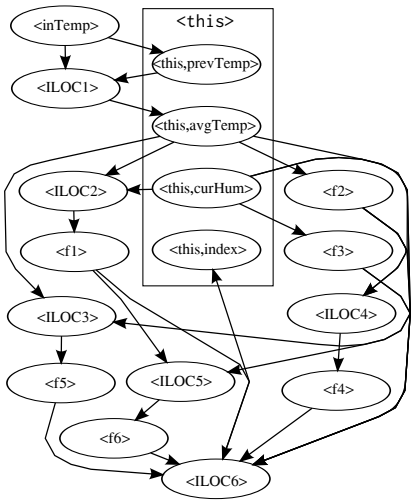

Fig. 3. Value Flow Graph for the Example

The value flow graph represents the following flow constraints that SInfer uses for type inference.

Flow Constraint: If there exists an explicit or implicit information flow from location n_1 to location n_2, then there must exist an edge $(n_1 \rightarrow n_2)$ that indicates the location type of n_1 must be higher than the location type of n_2.

We present the basic algorithm for generating the value flow graph below:

i) The algorithm first computes a mapping $\mathcal{R} \subseteq V \times T$ from variables to sets of location tuples. For each variable, the algorithm maps the variable to a location tuple that contains just the variable. For each field access, the algorithm maps the field access to a location tuple that consists of the reference variable followed by a sequence of fields.
ii) The algorithm also models implicit flows by using a stack $\mathcal{S}$ of sets of tuples. At each conditional branch, it pushes a set of conditional tuples onto the stack $\mathcal{S}$ and at each merge it pops the top set of tuples off the stack.
iii) The algorithm next adds edges for every flow to a variable or an object field. For example, it generates an edge $\mathcal{R}(\mathtt{y}) \rightarrow \mathcal{R}(\mathtt{x})$ for the assignment `x = y`. The algorithm also adds edges from the top set of tuples of the stack $\mathcal{S}$ to location tuples being assigned.

To illustrate the inference algorithm, we will use the example from Figure 2 with annotations removed. Figure 3 presents the value flow graph for the example. The nodes in the graph represent locations and the edges represent flows. For example, the edge from the node labeled <inTemp> to the node labeled <this,prevTemp> indicates that information may flow from the variable `inTemp` to a field with the `prevTemp` location of an object with the `this` location. We group nodes with the same composite location prefix together in a rectangle.

When an expression has more than one operand, our inference tool generates an extra node in the value flow graph, called an *intermediate node* as expressed by `ILOC` in Figure 3, which has incoming edges from the operand nodes and an outgoing edge to the original destination of the expression. It allows the SJava type checker to derive the GLB of the location types of the expression's operands.

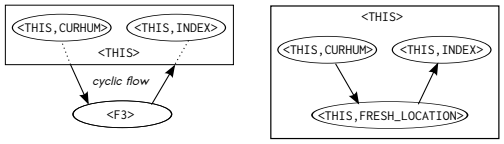

Fig. 4. Superfluous Cycle

B. Avoiding Unnecessary Cycles

As our algorithm proceeds, it decomposes the value flow graph into field and method hierarchies. In some cases, this decomposition can merge two different nodes in the value flow graph into the same node in a field or method hierarchy. If one of these nodes is reachable from the other, this merging has the potential to create a superfluous cycle in the lattice. These cycles can occur when either local variables, parameters, or objects are given imprecise method lattice types. We next discuss how our approach assigns more precise composite location types to each.

1) Local Variables: Within a single method, it is possible to introduce a superfluous cycle by assigning a method location type to each local variable (the default assignment strategy). Consider the following code excerpt:

```
1  public void calculateIndex(){
2    float f3 = c3 * this.curHum * this.curHum;
3    ...
4    this.index = f3 + ...;
5  }
```

The method `calculateIndex` takes a value from the field `curHum`, computes it, then stores the result back to the field `index`. If we assign the composite location ⟨`F3`⟩ to the local variable `f3`, it will introduce a cycle in the method lattice as illustrated on the left side of Figure 4. The problem is that the assignment in Line 2 implies that the method location ⟨`F3`⟩ is lower than the method location ⟨`THIS`⟩. By the same logic, the assignment in Line 4 would then create a cycle in the method location lattice. However, if we assign the variable `f3` to a composite location that begins with the method location `THIS`, e.g. ⟨`THIS`, `FRESH_LOCATION`⟩, we can avoid introducing any cycles as illustrated on the right side of Figure 4.

2) Parameters: Interprocedural value flows can also introduce a superfluous cycle in the location types corresponding to value flows across method calls. For example, consider the following code excerpt:

```
1 class foo {
2   int f,g;
3   void caller() {
4     int h = this.f;
5     callee(h);
6   }
7   void callee(int i) {
8     this.g = i;
9   }}
```

The method `caller` takes a value from the field `f` in Line 4 and passes it as the argument to the method `callee` in Line 5, and then the method `callee` stores the value of the corresponding parameter `i` back to another field `g` of the same object in Line 8. Suppose that the location types of the current object `this` are ⟨`THIS`⟩ for both methods. The assignment in Line 4 implies that the location of the variable `h` is lower than the location ⟨`THIS`, `F`⟩. However, if we assign the method location ⟨`LOCH`⟩, which is lower than the location ⟨`THIS`⟩ in the method lattice, to the variable `h` without considering a field location type, there is a cycle—the value of the location ⟨`LOCH`⟩ eventually flows into the location of another field of the same object ⟨`THIS`, `G`⟩ in Line 8 through the method invocation.

To address this problem, our analysis summarizes the callee's value flows that involve objects reachable from the caller's arguments and transfer them to the caller so that we can remove the cycle in a later stage. When a callee has a value flow with an object that is reachable from a parameter, the analysis adds the corresponding value flows to the value flow graph of the caller in terms of the arguments. Our interprocedural analysis is structured as a fixed point computation. Whenever a method flow graph changes, all methods that call that method are scheduled for reanalysis.

3) Objects: It is possible to inadvertently create a cycle in the graph where a single flow traverses a set of composite locations that do not all have the same prefix (e.g., ⟨`X`, `Y`⟩ in ⟨`X`, `Y`, `Z`⟩). Consider a location x in the value flow graph that (1) is reachable from a composite location y and (2) can reach a composite location z, where y and z share a common prefix that is not shared by x. The analysis constructs a new location that has the same prefix as y and z, followed by a fresh field location, and then assigns the new location to x.

To avoid a superfluous cycle, the process described here identifies one prefix that becomes the common prefix for all composite locations involved in the cycle. For example, in Figure 4, the algorithm identifies the common prefix `THIS` in the cycle, and then removes the cycle by assigning a new composite location with the prefix `THIS` to the location `F3`. However, it is possible for two objects to have two different cycles that each require assigning the other object to have the same prefix as a member of the first object. This kind of cycle is not representable by the SJava type system. A graph may have a cycle with more than one common prefix. For such cycles, there is a choice of which object to select to become the common prefix. Since we have not seen this case appear in practice, the SInfer implementation simply selects at random a class to become the common prefix and prints a message for the developer. If such cases do appear, the implementation could be modified to try different common prefixes.

4) Propagating Type Adjustments: The resolution of cycles globally updates locations. So the prefix adjustments must be propagated throughout the value flow graphs. This process begins in the main event loop and proceeds downwards to each leaf method in the call graph. The composite location of a method call argument is propagated from the caller to the callee by translating the first element of the argument's composite location into the callee context. For example, suppose there is a function call in which: (1) the argument has the composite location ⟨`OBJ`, `A`, `B`⟩, (2) the object whose method is invoked has the location ⟨`OBJ`⟩ in the caller, and (3) the callee defines the location of the current object `this` as ⟨`THIS`⟩. In this case, the prefix ⟨`OBJ`⟩ in the caller is translated into the new prefix ⟨`THIS`⟩ in the callee, resulting in a composite location ⟨`THIS`, `A`, `B`⟩ for the method parameter within the callee.

C. Inferring Program Counter Locations

The program counter may be annotated with a location type, and if no annotation is provided, then it defaults to the top location. But this default assignment may be higher than

the program counter location at a method call. In this case the type checker will produce a type error at the call site.

Computing the lowest valid PC location provides the most flexibility as the method can be used whenever the caller's program counter location is higher than the callee's declared program counter location. Our tool first computes the set of parameter nodes P_{in} that have incoming flows in the flow graph. It then generates a program counter node PC satisfying the following constraint in the value flow graph.

Program Counter Constraint: For each parameter node $p \in P_{in}$, there must exist an edge $(\texttt{PC} \rightarrow p)$ *that indicates that the location type of the node PC must be higher than the location type of the parameter p.*

When one of the parameter locations $p \in P_{in}$ is higher than all the others and it has a field location type, the node PC is assigned a new composite location with a field location type that is higher than the highest parameter node. If all the parameters have incoming flows, the tool elides the program counter annotation and simply relies on the default annotation.

D. Return Values

Annotating the return type of a method is necessary for completeness. When a program is type-checked, SJava computes the caller location for the return value that is consistent with the location types of the parameters and the return value. Therefore, computing the highest location for return values provides more flexibility to the caller context.

Our tool first computes the set of return value nodes $r \in R$ such that there is a path $(p \rightsquigarrow r)$ in the value flow graph from a parameter node p to a return value node r. Then, it generates a return location node RLOC satisfying the following constraint for return values.

Return Location Constraint: For each return value node $r \in R$, there must exist an edge $(r \rightarrow \texttt{RLOC})$ *that indicates that the location type of node* RLOC *must be lower than the location type of the return value node r.*

When one of return value nodes in the set R is lower than all of the others and it contains a field location type, the node RLOC is assigned a new composite location with a field location type relative to the lowest return value node, which provides more flexibility at call sites.

E. Hierarchy Graph

The goal of this phase in the algorithm is to transform the value flow graph into the method and field hierarchy graphs, each of which later is converted to a lattice suitable for producing annotations. In the value flow graph, information flows are not modeled in a modular manner — edges correspond to flows in both field and method lattices. Therefore, SInfer decomposes the value flow graph into hierarchy graphs to capture information flows at the level of individual classes and methods, which makes the entire system composable.

We first classify edges in the value flow graph into two types of flows:

- Method flows: An edge represents a method flow if the first elements of two nodes are not identical. For example, the edge from $n_1 = \langle v_1, \texttt{f}_m, ..., \texttt{f}_n \rangle$ to $n_2 = \langle v_2, \texttt{f}_p, ..., \texttt{f}_q \rangle$ in the value flow graph represents a method flow from v_1 to v_2 in the corresponding method.
- Field flows: An edge represents a field flow if the prefixes of two nodes are identical and the subsequent fields are not identical. For example, the edge from $n_1 = \langle v_1, ..., \texttt{f}_m, \texttt{f}_n \rangle$ to $n_2 = \langle v_1, ..., \texttt{f}_m, \texttt{f}_o \rangle$ in the value flow graph represents a field flow from $\texttt{f}_n$ to $\texttt{f}_o$ in the corresponding class.

The inference algorithm next decomposes value flow graphs into two types of hierarchy graphs: a *method hierarchy graph* and a *field hierarchy graph*. For each method flow in the value flow graph, a corresponding edge is generated in the method hierarchy graph. For each field flow in the value flow graph, a corresponding edge is generated in the field hierarchy graph of the class containing the fields.

Definition 2. *(Hierarchy Graph) A hierarchy graph is a directed graph $G = (H, E_h)$, where a node corresponds to either a field in the field hierarchy or a local variable in the method hierarchy, and an edge $(h_1 \rightarrow h_2) \in E_h$ denotes that there is either a field flow in the field hierarchy or a method flow in the method hierarchy from h_1 to h_2.*

We present the basic algorithm for translating the value flow graph into method and field hierarchy graphs below:

i) The algorithm first compares the source composite location `srcloc` and destination composite location `dstloc` and computes the index `idx` of the first difference in the two composite locations. We will refer to the corresponding hierarchy as `diffhierarchy`.

ii) The algorithm next checks whether adding an edge from `srcloc[idx]` to `dstloc[idx]` in the hierarchy `diffhierarchy` will create a cycle.

iii) If the addition would introduce a cycle, the algorithm merges all nodes in the cycle into a single shared location.

iv) Otherwise, the algorithm adds an edge from `srcloc[idx]` to `dstlock[idx]` in the hierarchy `diffhierarchy`.

The algorithm translates the edges in the value flow graph into the equivalent flow edges in either a method or field hierarchy. Step one of the algorithm identifies the first element of the two composite locations that differ. Since the SJava type checker will evaluate the composite locations in lexical order, the algorithm must generate location flows that adhere to lexical ordering. Adding a flow in the corresponding hierarchy guarantees that the resulting location flows will type check. It is possible that adding the edge could introduce a cycle into a method or field hierarchy graph. If this occurs, the algorithm eliminates the cycle by merging all of the locations into a shared location. Figure 5 shows the method hierarchy graph for the `calculateIndex()` method and Figure 6 shows the field hierarchy graph for the `Weather` class.

F. Converting the Hierarchy Graphs into Lattices

At this point, the hierarchy graphs capture flows between all locations in the program. While the hierarchy graphs are partial orders, they may not be lattices as the GLB and LUB are not necessarily well defined. We may need to insert extra nodes into the partial order to make the GLB and LUB well defined. The problem of finding the smallest complete lattice that contains a partial order is known as the Dedekind-MacNeille completion. We use the algorithm developed by Nourine and

Fig. 5. Method Hierarchy

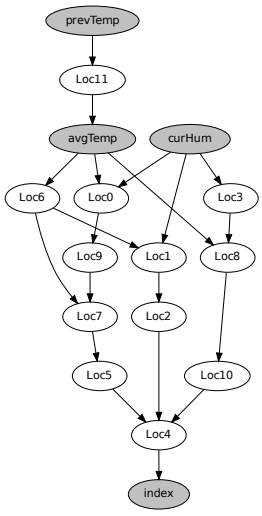

Fig. 6. Field Hierarchy

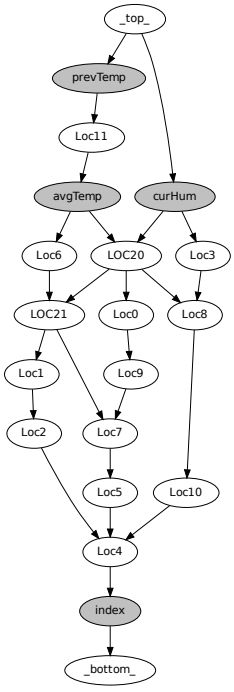

Fig. 7. Field Lattice

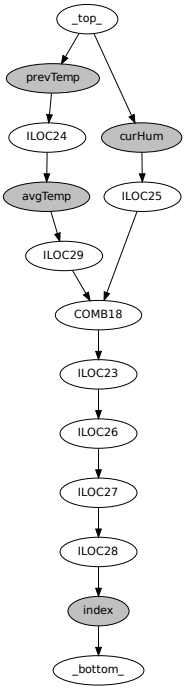

Fig. 8. Simplified Field Lattice

Raynaud [16] to compute the Dedekind-MacNeille completion of the hierarchy graphs to generate lattices. Figure 7 shows the inferred field lattice of the `Weather` class.

G. Non-self-stabilizing Programs

A program can fail to self-stabilize if either (1) bad values remain at a location indefinitely or (2) bad values cycle indefinitely in a cyclic value flow. When values are not evicted from a location, SInfer may infer location types that type check, but SJava's static eviction analysis will reject the program. In the case of a cyclic value flow, SInfer will attempt to use a shared location to eliminate the cycle. If the cycle contains either object references or fields of different objects, SInfer will abort because it cannot be represented in the SJava type system. In this case, SInfer provides value flow graphs to help developers fix the problems. For cycles that can be represented using shared types, SInfer may potentially infer type annotations that type check. However, the stronger static eviction criteria required for shared locations will cause SJava's static eviction analysis to reject the program.

H. Discussion

Our initial approach captured flow constraints using value flow graphs and hierarchy graphs, then produced lattices which are precisely compatible with the flow of values in the program. To maintain maximum precision, our algorithm created a unique mapping of each location type onto each node in the lattice. However, in practice, a program often defines a large number of variables and information paths. Therefore, maintaining maximally precise flow information may lead to a very complicated lattice that is incomprehensible and infeasible for developers to maintain. For example, although the lattice in Figure 7 does not look very complicated, it is a representation of value flows produced by the relatively small program. After applying our initial implementation to the benchmarks, we found that the resulting lattices were too complicated for practical use. For example, the lattice for the `SynthesisFilter` class of the MP3 Decoder benchmark defines 997 locations types and has 10,491,169 possible information paths from the top to the bottom in the lattice. This generated lattice is clearly incomprehensible to developers.

V. Simplification

The inference algorithm described in the previous section infers unique location types for every variable and field declaration, but often produces overly complicated lattices. However, generating simple lattices by reusing existing location types whenever possible can sometimes limit reusability. The reason is imprecise ordering relations between parameters and fields may require significant modifications when using the methods or fields in new contexts.

Our goal is to find a reasonable tradeoff between precisely modeling flow constraints and the simplicity of the resulting lattices. The idea is to maintain precise ordering relations between interface members (e.g., parameters, return values, program counters and fields) while reusing location types for non-interface members (e.g., local variables).

A. Constructing Interface Hierarchy Graphs

SInfer optimizes for precise modeling of interface members by generating *interface hierarchy graphs* which only contain interface members. The process of simplification begins with removing non-interface nodes from the hierarchy graphs and then patching edges across the removals (potentially generating multiple edges for a removed edge). Returning to the example, the `Weather` class has four fields as interface members, shown as shaded nodes in Figure 6. Figure 10 shows the interface hierarchy graph for the `Weather` class.

B. Simplifying the Interface Hierarchy Graph

SInfer next simplifies the interface hierarchy graphs by identifying nodes to merge. If two nodes have incoming edges that originate from the same set of nodes and outgoing edges that target the same set of nodes, then those two nodes can be merged while still maintaining the same precision (merging the nodes does not allow any new information flows). SInfer identifies and merges such nodes.

It is possible for method and field hierarchies to contain redundant edges. If a hierarchy contains the edge $e = (n \rightarrow n')$ where n' is reachable from n through a path that does not contain the edge e, then e is redundant. SInfer identifies and removes redundant edges in the hierarchies.

C. Inserting Merge Points

At this point, all nodes in the simplified hierarchy graph are either fields (for field hierarchies) or parameters, return values, and program counter locations (for method hierarchies). However, when the program combines value flows from more than one interface node, it needs a node in the hierarchy to store the combined flows. Consider the interface hierarchy graph shown in Figure 9. This hierarchy shows how information from the `a`, `b`, `c`, and `d` fields are used to compute the values stored in the `f` and `g` fields. Such a computation could potentially begin by combining information from the `b` and `c` fields and storing it into a local variable. There is no location type in the hierarchy as shown that can be used for such a local variable. If we used

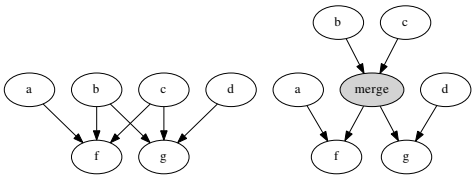

Fig. 9. Merge Point

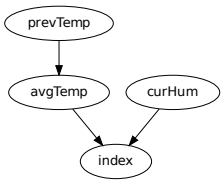

Fig. 10. Field Interface Hierarchy

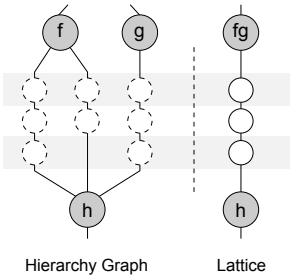

Fig. 11. Hierarchy Graph and Lattice

the location type of either field `f` or `g`, it would introduce a new flow into the hierarchy. Therefore, our tool must insert a merge point location type in the hierarchy that combines the information from fields `b` and `c` but is above both fields `f` and `g`. SInfer inserts a merge point whenever a non-interface node in the hierarchy graph combines incoming flows that originate from more than one interface node.

D. Converting the Interface Hierarchy Graphs into Lattices

The field and method hierarchy graphs currently capture the information flows between field and method interfaces, respectively. We compute the Dedekind-MacNeille completion of the field and method hierarchy graphs to generate field and method interface lattices, respectively.

E. Inserting Local Variable Locations

At this point, the interface lattices precisely capture information flows at the method or field interfaces. Now we insert locations for local variables into the lattices[3]. The idea is to splice in local variable nodes along existing edges in the interface lattice without changing the lattice's meet or join structure. In the hierarchy graph, there could be more than one path consisting of local variable nodes between two interface nodes that are directly connected with each other in the interface hierarchy graph. These paths are comprised of two types of local variable nodes: normal nodes and shared nodes. The problem is to find the shortest sequence of nodes that contains every ordering sequence in a given set of paths.

This problem is a generalization of the Shortest Common Supersequence problem[4], which is known to be NP-complete [18]. We implemented the following approach that maps local nodes in the hierarchy graph to the location types in the lattices, heuristically minimizing the number of location types in the lattice.

i) For a local variable node l in the hierarchy graph, find the lowest node m in the interface lattice that is above l.

ii) Count the number of hops d between the local variable node l and the node m. Each hop corresponds to a pair of a normal node and a shared node. SInfer will insert either type of node to a pair if needed.

iii) We next convert the node m into a chain of nodes with the last node in the chain having m's outgoing edges. SInfer first checks if there already exists the dth hop pair along this chain. If such a pair does not exist or the same type of node does not exist in the pair, we insert a new node. Otherwise, we reuse the existing node.

SInfer generates a final lattice that admits more flows between local variables than the actual program performs, but suffices to show that the program self-stabilizes.

Figure 11 illustrates how our approach optimizes the hierarchy graphs. In the figure, the shaded circles represent the interface nodes and the dotted circles represent the local variables. First, the analysis simplifies the graph by merging two interface nodes `f` and `g` since both nodes share the same incoming and outgoing edges in the interface hierarchy graph. Then, it generates the lattice in which local variables share locations. Figure 8 presents the inferred lattices for the example, which is much simpler than the non-optimized lattice in Figure 7. In the lattice, local variables with the same height are assigned to the same node, and non-interface members (white nodes) are arranged in a simple structure between interface members (shaded nodes) while the optimization maintains precise orderings between interface members.

VI. Evaluation

We have implemented our inference tool in the SJava compiler and evaluated it on an Ubuntu Linux 12.04 machine with an Intel Core i7 3770 CPU. We have evaluated the tool by inferring annotations for three SJava applications: JLayer, an MP3 decoder; LEA, an eye-tracker; and Sumo Robot, a robot controller. Our implementation and benchmarks are available for download at http://demsky.eecs.uci.edu/compiler.php.

A. Methodology

We evaluated SInfer by taking the source code for the three benchmarks and inferring annotations. As the original applications required minor changes to make them self-stabilizing, we took the modified versions of the SJava benchmark and removed all of the location type annotations.

Correctness We used the SJava type checker to verify the correctness of the generated annotations. All three benchmarks type check and pass SJava's eviction analysis, and thus they are self-stabilizing.

Simplification Goals Our evaluation was designed to evaluate the usefulness and understandability of the inferred lattices. Even though such evaluations depend on subjective criteria and how developers use the results, the complexity of the lattices is an important factor in evaluating the usefulness and understandability. Therefore, we developed two metrics to measure the complexity of the lattices. First, we approximated the complexity by comparing the total number of locations in the lattices between SInfer and the naïve approach that attempts to maintain maximal precision described in Section IV. However, measuring the complexity is more complicated than comparing the total number of locations. For example, if a large number

[3]Field lattices may contain locations for local variables with composite locations.

[4]A shortest common supersequence problem can be represented as our minimization problem by creating a unique path for each input sequence, encoding 0 with a normal node and 1 with a shared node. Thus finding an optimal solution to our problem is at least as difficult as the shortest common supersequence problem.

Benchmark		Simple(<=5)		Complex(>5)		Time	LOC
		Locations	Paths	Locations	Paths		
MP3	manual	141	35	268	48	n/a	15,634
	naïve	176	61	1,998	294,624,128	10.92s	
	SInfer	205	62	421	542	12.15s	
Eye	manual	215	59	69	12	n/a	4,571
	naïve	183	58	503	905	0.47s	
	SInfer	161	67	343	42	0.51s	
Robot	manual	132	36	80	14	n/a	3,201
	naïve	149	44	161	152	0.18s	
	SInfer	161	45	79	18	0.20s	

TABLE 1. EVALUATION RESULTS

of locations are arranged into a single line, this structure could be more easily understandable than a complex structure with a smaller number of locations. Therefore, we defined another quantitative measurement, the number of paths from the top to the bottom in a lattice, which captures the number of different ways values flow through a lattice. McCabe [14] developed a similar metric to measure program complexity.

Table 1 provides information about the lattices generated by the naïve approach and SInfer. It also shows information about the manual annotations. Even though SInfer generates a smaller number of locations and paths than the naïve approach, we found that the total numbers tend to be biased by many simple lattices. Therefore, we split lattices into two categories: simple lattices and complex lattices with complex lattices defined as having more than 5 nodes. The threshold columns labeled <= 5 and > 5 show the total numbers of the simple lattices and the complex lattices respectively. Note that, in some cases, the total number of SInfer lattices in the simple category is larger than the total number of the naïve approach because SInfer is able to simplify complicated lattices and place them in the simple category. The last two columns show the time for type inference and lines of code for each benchmark. SInfer is slower than the naïve approach because the former requires an additional process for the simplification.

B. MP3 Decoder

JLayer is an MP3 decoder and is available at http://www.javazoom.net/javalayer/javalayer.html. The decoder self-stabilizes because it flushes out all non-loop invariant state within a bounded number of frames. After an error occurs, if the event loop continually retrieves new audio frames, it will eventually resume the normal behavior from an arbitrary state.

Of the three benchmark applications, the MP3 decoder had the highest potential for complicated annotations. Many methods in the program employ a processing pattern which first loads data into a set of source fields, then transforms the data in a sequential computation using several local variables, and finally store the results into a set of destination fields. The problem is that the multiple stages of computation that extensively use temporal variables to store intermediate results create a large number of value flows.

To quantify how well we have met the simplification goal, we present results comparing SInfer to the manual annotations and to the naïve approach in Table 1. It is clear that the new strategy helps effectively reduce the number of locations and paths. For the MP3 decoder, SInfer generated 421 location types and 542 paths for complex lattices, whereas the naïve approach generated a total of 1,998 location types and 294,624,128 paths. SInfer infers slightly more complicated annotations than the manual annotations. The manual approach used some tricks to reduce location types — the manual annotations used shared location types simply to avoid generating a chain of location types in a lattice. While this approach does reduce the number of annotations that must be written, it can also be misleading as it may lead new developers to believe that a cyclic value flow exists where it does not. Thus although the automatic annotations contain more location types, they may in fact be preferable for program understanding.

Ideally, SInfer will generate annotations that the developer can intuitively understand. The MP3 decoder is our best benchmark for evaluating the readability of the generated annotations, because it is the most complex. We manually examined the generated lattices and found that their structures clearly show the flow of values through the program. In the lattices, the location types assigned to the fields outlined a distinct hierarchy, and it was easy to correlate each level of that hierarchy with a phase of the sequential decoding process. This provides some evidence that the tool may be useful for program understanding and debugging.

C. Eye Tracking

LEA is an eye tracking library and is available at http://sourceforge.net/projects/lea-eyetracking/. Every event loop iteration takes an input image from a web cam, tracks eye movement, and returns a direction. LEA maintains up to the last three eye positions to derive the movement direction. LEA self-stabilizes because it flushes all non-loop invariant values after the three loop iterations.

The original manual annotations identified an opportunity to avoid introducing a large number of composite locations: the part of the computation that detects eye positions could use the same location for all eye position fields. Our inference tool identifies the same opportunity to simplify the lattice, and generates lattices that are straightforward to understand.

D. Robot Controller

Sumo Robot is a Java library for developing robot controllers, available at http://java.net/projects/sumorobots/. The main controller is self-stabilizing because it does not maintain persistent state and overwrites all control variables at every iteration. Each iteration of the event loop reads data from the sensors, selects a movement type and speed, and generates a motor controller command.

The program follows a common pattern that is likely shared by a broad range of embedded controllers: at every iteration, such a controller takes a new input, processes it, overwrites all execution-specific memory locations, and emits a result. This pattern results in a straightforward location type hierarchy, because (1) each computation stage directly matches one level of the type hierarchy and (2) Sumo Robot does not rely heavily on the use of objects, which would complicate the hierarchy.

VII. RELATED WORK

Self-stabilization was initially developed as an approach to build distributed algorithms that were robust to failures [3]. Dolev *et al*. [5], [6], [4] recently developed an approach that ensures that the underlying layers (processor, operating system, and compiler) preserve the self-stabilizing nature of

an application. This earlier work does not check that the actual application is self-stabilizing. SJava is a type system and static analyses that together check that a Java program is self-stabilizing[8]. A barrier to the adoption of this work is that the technique requires additional type annotations. This paper develops a tool that can infer an initial set of type annotations for the developer — reducing the need for manual annotation.

Researchers have developed type systems for language-based information flow to check that an application's information flows do not violate the desired requirements for security and energy savings[15], [20], [21]. We believe that our work in inferring flow annotations can be adapted for other information flow type systems. A number of approaches have been proposed for secure information flow inference. Smith *et al*. [23] and King *et al*. [10] attempt to reduce the annotation burden, but they still require developers to define security policies. SInfer automatically infers all specifications and type annotations. Vaughan *et al*. [24] present a security policy inference tool which does not infer method annotations. Livshits *et al*. [12] develop a probabilistic inference tool for explicit information flow. Their tool only infers information flows in terms of the methods involved, which is too coarse for the SJava type system. The constraint-based type inference system[17] has a different goal from SInfer. Whereas they try to infer types as precisely as possible, we infer location types that meet our simplicity goal while losing some precision.

A number of tools have been developed to infer specifications. One prominent example is Daikon [9], which infers specifications for program variables and fields. Other tools have been used to automatically infer Java generics annotations [7] and concurrency specifications [1]. While our work shares the high-level goal of alleviating the manual annotation labor, the techniques and goals of these tools differ.

Failure-oblivious computing [19] enables programs to continue execution past memory errors by manufacturing values for reads or discarding writes. Other work detects bugs and tries re-execution in a slightly different environment [22]. Data structure repair [2] takes an interventional approach; upon detecting data structure corruption, it repairs them with respect to a specification. Data structure repair only guarantees that a program will reach some consistent state, while self-stabilization guarantees that all effects of the bug will eventually disappear. Moreover, self-stabilization does not require a specification and therefore eliminates the need to precisely define correct behavior.

VIII. Conclusion

Bugs have long plagued software systems. While self-stabilization has been used as an approach for building robust distributed systems, automatically checking that code self-stabilizes is relatively new. In certain important domains, self-stabilization can improve the robustness of software applications. Moreover, self-stabilization is complimentary to existing approaches for improving reliability.

A barrier to the widespread adoption of checking self-stabilization is that the existing approach requires manual annotations. This paper presents an approach for automatically inferring these annotations. Our experience indicates that SInfer can successfully infer annotations for our benchmarks.

Acknowledgment

This project was partly supported by the National Science Foundation under grants CCF-0846195 and CCF-1217854.

References

[1] J. Burnim and K. Sen. DETERMIN: Inferring likely deterministic specifications of multithreaded programs. In *ICSE*, 2010.

[2] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *ICSE*, 2005.

[3] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *CACM*, 17, 1974.

[4] S. Dolev, Y. Haviv, and M. Sagiv. Self-stabilization preserving compiler. *TOPLAS*, 31, 2009.

[5] S. Dolev and Y. A. Haviv. Self-stabilizing microprocessor: Analyzing and overcoming soft errors. *TC*, 55, 2006.

[6] S. Dolev and R. Yagel. Toward self-stabilizing operating systems. In *DEXA*, 2004.

[7] A. Donovan, A. Kieżun, M. S. Tschantz, and M. D. Ernst. Converting Java programs to use generic libraries. In *OOPSLA*, 2004.

[8] Y. Eom and B. Demsky. Self-stabilizing Java. In *PLDI*, 2012.

[9] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE*, 2000.

[10] D. King, T. Jaeger, S. Jha, and S. A. Seshia. Effective blame for information-flow violations. In *FSE*, 2008.

[11] Y. Liu and A. Milanova. Static analysis for inference of explicit information flow. In *PASTE*, 2008.

[12] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification inference for explicit information flow problems. In *PLDI*, 2009.

[13] H. M. MacNeille. Partial ordered sets. *Transactions of the American Mathematical Society*, 42(3), 1937.

[14] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4), 1976.

[15] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, 1999.

[16] L. Nourine and O. Raynaud. A fast algorithm for building lattices. *Information Processing Letters*, 71(5-6), 1999.

[17] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA*, 1994.

[18] K.-J. Räihä and E. Ukkonen. The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science*, 16(2), 1981.

[19] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, 2004.

[20] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[21] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI*, 2011.

[22] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. In *ATC*, 2005.

[23] S. F. Smith and M. Thober. Improving usability of information flow security in Java. In *PLAS*, 2007.

[24] J. Vaughan and S. Chong. Inference of expressive declassification policies. In *IEEE Symposium on Security and Privacy*, 2011.

[25] P. Wadler. Linear types can change the world! In *PROCOMET*, 1990.