Inferring SIL Specifications with the Abstract Interpreter Sample

Severin Heiniger

Master's Thesis Report

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

April 6, 2014

Supervised by:
Dr. Alexander J. Summers
Milos Novacek
Prof. Dr. Peter Müller

inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Automated verifiers can prove the correctness of programs with respect to their specifications. However, writing such specifications is cumbersome and often requires considerable level of expertise. In this thesis, we present a novel approach to automatically inferring specifications for programs written in SIL, an intermediate language with a permission-based verification methodology. We focus on programs that operate on recursive data structures such as linked lists and trees, whose specifications require non-trivial recursive assertions. Our approach builds upon a recent combined heap and value analysis based on abstract interpretation. However, the results of the heap analysis are not directly amenable for extracting the specifications. Hence, we refine the analysis with additional information to enable the extraction of specifications. We have implemented our approach in the static analyzer Sample. The use of a static analysis for inferring specifications rather than just program properties is an intriguing direction of research that, to the best of our knowledge, has not been widely researched. Experimental results show that our approach can infer specifications for programs that operate on recursive data structures, without requiring any user-provided annotations.

Contents

1 Introduction

Software permeates our world more than ever. In cases such as transportation systems and medical equipment, human lives depend on software. It is of paramount importance to be able to guarantee the correctness of such critical software.

Guaranteeing the correctness of a program via testing is impossible, as the program could take infinitely many inputs. A solution is to formally specify a program's behavior and mathematically prove that the program satisfies this specification, without actually running the program. Constructing such mathematical proofs by hand is time-consuming and often infeasible. Hence, several automated verifiers have been developed. A verifier will either succeed in proving that a program satisfies its specification, or it will output a list of errors that may indicate a bug in the program or that the specification is incomplete, that is, lacks certain information required for the proof.

While verification is automated, writing specifications for programs is time-consuming. Our thesis aims to reduce the overhead required to specify programs written in a verification language called SIL, by automatically inferring specifications from the programs themselves. In order to infer SIL specifications, we extend a recent combined heap and value analysis [5].

Section 1.1 provides an overview on how SIL programs are specified and Section 1.2 briefly describes the heap analysis that our project is based on. Section 1.3 then provides an outline of our approach to inferring specificiations for SIL programs.

1.1 Automated Software Verification with SIL

The Semper Intermediate Language (SIL) is an object-based verification language influenced by Boogie [10] and Chalice. SIL programs contain specifications in the form of method pre- and post-conditions as well as loop invariants.

SIL is a core part of the Semper project at the Chair of Programming Methodology at ETH Zurich that aims to build an automated verification infrastructure for concurrent programs written in modern programming languages such as Scala. The Semper project includes two SIL verifiers[1]. Currently, there are front-ends that can translate Scala [1] and Chalice [9] programs into SIL programs for verification.

1.1.1 Framing

One of the reasons why statically verifying object-based programs is challenging is that a verifier needs to know what parts of the heap a method call is guaranteed to leave unchanged. This is called the *framing* problem.

SIL solves this problem by using *implicit dynamic frames* [14]. SIL uses *permissions* to express which heap locations a method may access. Permissions exist per heap location

[1] Carbon [7] and Silicon [13], based on verification condition generation and symbolic execution, respectively.

and methods can gain or lose permissions at various points.

For example, a method has permission to all the fields of any object it allocates. The method may only access data structures supplied as parameters if the precondition includes corresponding permissions. Any caller of the method needs to supply these permissions. Similarly, postconditions allow methods to return permissions to their caller.

1.1.2 Abstract Predicates

Many interesting programs operate on recursive heap data structures, such as linked lists that consist of an unbounded number of heap locations. In SIL, recursive abstract predicates [11] are the key to specifying such programs. An abstract predicate abstracts over heap locations and can express the permission to (and other properties of) a whole recursive data structure.

Example. Listing 1 depicts a recursive predicate that specifies a linked list. The predicate contains the permission to the `val` and `next` field of the given list element `list`, and also contains the permission to all remaining elements recursively via `next`.

```
predicate valid(list: Ref) {
  acc(list.next) && acc(list.val) &&
  (list.next != null ==> acc(valid(list.next)))
}
```

Listing 1: Recursive predicate that specifies a linked list.

Folding and Unfolding Predicate Instances. SIL verifiers distinguish between holding an *instance* of a predicate via its name, and holding its body. To obtain the permissions directly mentioned in the body of a predicate, the SIL method needs to tell the verifier with an **`unfold`** statement to replace the predicate instance with its body. A **`fold`** statement performs the inverse operation.

1.2 Static Program Analyzer Sample

The Semper project includes a static program analyzer called Sample (Static Analyzer of Multiple Programming LanguagEs) [4]. Sample is based on the abstract interpretation framework [2, 3], to soundly approximate possible states of a program.

Sample contains a combined heap and value analysis [5] that infers invariants for programs that operate on recursive data structures such as linked lists and trees. For example, an inferred invariant may express that the shape of the heap is a linked list and that the values in that list are sorted. The analysis does not require any program annotations.

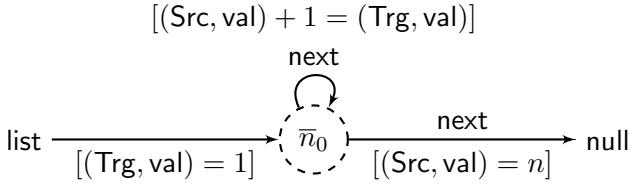


Figure 1: Abtract heap that represents a linked list of the natural numbers 1 to n.

1.2.1 Abstract Heaps

At every point in the program, the heap analysis finds an abstract heap that represents every possible heap that could exist at that point when running the program.

The analysis represents such an abstract heap as a directed graph, where nodes represent one or more objects at runtime. Edges between nodes represent possible field references between these objects. The heap analysis tracks the relationships among value fields and variables as an abstract value state on each edge.

1.2.2 Connection to Recursive SIL Definitions

The heap analysis is agnostic of concepts such as permissions and predicates that are part of the SIL verification methodology. However, the heap analysis can infer abstract heaps that represent recursive data structures. The representation with summary nodes suggests a connection to recursive assertions such as abstract predicates in SIL.

The abstract heap in Figure 1 represents a data structure that recurses over next fields and is terminated by null. Such a structure can be described by the abstract predicate from Listing 1.

1.3 Overview

This project introduces a novel approach of inferring specifications for SIL methods via extending Sample's combined heap and value analysis.

The project focuses on methods that operate on recursive data structures such as linked lists and trees. Inferring specifications for such methods is challenging. It is necessary to identify predicates that specify the data structures, and which instances of these predicates need to be part of the method's precondition, postcondition and loop invariants. The method may also require `fold` and `unfold` statements at certain points.

The combined heap and value analysis can analyze methods that operate on recursive data structures. However, its representation of recursive data structures is not amenable for identifying recursive predicates that specify them.

We have devised a way to extend the heap analysis such that the analysis results contain sufficient information to extract specifications. Our extended analysis incrementally builds up information about predicates and predicate instances, based on how the

method operates on the heap. Our analysis exploits the information it has already learned about predicates to make the heap analysis more precise. Concretely, the analysis exploits that recursive predicates constrain the possible shapes of the data structures they specify.

Once a method is analyzed, we translate abstract heaps into SIL assertions such as method preconditions and postconditions as well as loop invariants and inject them into the original SIL program. For example, the resulting abstract state at the end of the method can be translated into a method postcondition. We also rely on information accumulated during the analysis to add `fold` and `unfold` statements to the method.

The SIL program with inferred specification can be presented to the user or directly supplied to a verifier for verification. The inferred specification may not be sufficient for a successful verification, but may still provide valuable information to the user and serve as a starting point for further refinement of the specification.

Figure 2 depicts the steps of our approach explained above. We will describe each of these steps in this report, with focus on the analysis.

1.4 Outline

The report is structured as follows: Sections 2 and 3 provide further background on SIL the combined heap and value analysis, respectively. Section 4 describes how SIL programs are translated such that Sample can analyze them. Section 5 discusses challenges associated with inferring specifications for SIL programs. While Section 6 describes our analysis and how it addresses these challenges, Section 7 explains how to extract SIL specifications from the results of that analysis. Section 8 covers interesting technical work conducted as a part of the project. Section 9 evaluates the capabilities and limitations of our approach. Finally, Section 10 concludes the report.

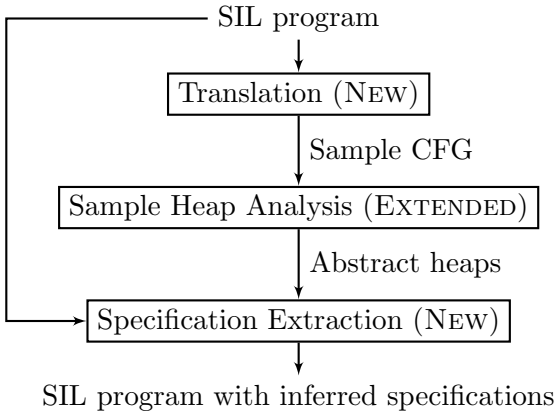


Figure 2: Flow of information for inferring specifications for a SIL program.

2 The Semper Intermediate Programming Language (SIL)

The following section builds up on the introduction to SIL in Section 1.1 and provides further details on the language. Section A provides the syntax of SIL.

2.1 Types and Objects on the Heap

SIL allows programs to allocate and use objects with fields on the heap. However, an important difference to real-world object oriented languages is that there are no classes. All objects have the same type `Ref`. SIL front-ends ensure themselves that a field access only occurs on an object that provides such a field. This restriction makes SIL verifiers easier to implement.

Besides reference types, there is also a boolean (`Bool`) and an integer type (`Int`).

2.2 Framing

Recall that SIL uses permissions to express which heap locations a method may access. The permission to a heap location represented by the field `f` of an object `o` is denoted as a *field access predicate* of the form `acc(o.f)`.

A method may only access a heap location such as `o.f` in a state where it holds to the permission to that heap location. This restrictions also applies to expressions in the specification that access heap locations. Expressions must be *framed* by the corresponding permissions, meaning that the permissions must be held in the state in which the expression is evaluated. An assertion is called *self-framing* if it requires permission to all heap locations that the expressions in the assertion access. For example, the assertion `acc(o.f) && o.f == 0` is self-framing, but the assertion `o.f == 0` is not. SIL requires that all assertions are self-framing.

Permission Transfer. When creating an object, the method automatically holds the permission for each of the fields of the object. The method `client` in Listing 2 contains an object creation, and holds the permission to perform the subsequent field assignments.

In other cases, the method needs to somehow obtain the permission. For example, the method `setToZero` in Listing 2 needs permission to the field `f` of its argument `o`. By adding the field access predicate `acc(o.f)` to the method's precondition, the verifier can assume that the method holds the permission to `o.f` at the beginning of the method. In order to call a method that requires a certain permission, the caller needs to hold that permission and give it up (*exhale* it).

Since the caller needs to give up the permission, methods usually return permissions to the caller, which can be achieved with access predicates in the method's postcondition. After the method call, the caller obtains (*inhales*) all permissions in the postcondition.

```
var f: Int

method setToZero(o: Ref)
   requires acc(o.f)
   ensures acc(o.f) && o.f == 0 {
   o.f := 0
}

method client() {
   var o: Ref
   o := new()
   o.f := 42
   setToZero(o)
   assert acc(o.f) // Still have permission to o.f
}
```

Listing 2: SIL program that illustrates the transfer of permissions.

For example, the method `setToZero` in Listing 2 returns the the permission to the location `o.f` to its caller. If it did not, the caller would not be allowed to access `o.f` any more after the method call. The permission would have been lost.

There are restrictions on how access predicates can be used in assertions. For example, they cannot be negated, occur in disjunctions or the left-hand side of implications.

2.3 Abstract Predicates

As explained in Section 1.1.2, recursive predicates make it possible to specify unbounded recursive data structures. The predicate `valid` in Listing 3 specifies a linked list recursing over the field `next`.

```
predicate valid(list: Ref) {
  acc(list.next) && acc(list.val) &&
  (list.next != null ==> acc(valid(list.next)))
}
```

Listing 3: Recursive predicate that specifies a linked list.

SIL uses `acc(valid(o))` to denote an instance of predicate `valid` at the object `o`. The instance contains the permission to all `val` and `next` fields reachable from the object `o` via `next`.

The permissions in the predicate instance are only available to the verifier after it unfolds the predicate instance. The statement `unfold acc(valid(o))` replaces the instance with the body of the predicate. That is, the method has permission to the fields `o.next` and `o.val` after unfolding. A `fold acc(valid(o))` statement performs the inverse operation. These statements are *ghost* statements. They are only meaningful in the context of verification and do not affect the execution of the program.

In addition to abstract predicates, SIL also offers abstraction functions that make it possible to specify recursive properties. In this project, we only consider abstract predicates for specification.

Acyclicity. The definition of the predicate `valid` guarantees that linked lists are acyclic. In SIL, it is impossible to hold a cyclic predicate instance [15]. That is, unfolding a predicate instance recursively never yields the same predicate instance again.

3 Combined Heap and Value Analysis with Sample

In this section, we give a brief overview of how abstract interpretation works and describe the combined heap and value analysis used for our project.

3.1 Abstract Interpretation

Programs perform computations in some concrete domain of values with each statement having a concrete semantics. In general, a program's state space at a given program point is potentially infinite, making it intractable for the program analysis to compute it.

However, the concrete domain and semantics can be approximated by an abstract domain and semantics that is computable and still preserves the properties of interest. Abstract interpretation [2, 3] is a framework that guarantees a sound approximation of the semantics of programs and is used for reasoning about the program correctness.

3.1.1 Forward Analysis

A common static analysis of a program aims to precisely approximate the abstract state of a program at any given program point. A forward analysis starts from some initial abstract state at the beginning of the program and for each program point computes the abstract post-state of the statement at that program point by applying the abstract semantics to the abstract pre-state of the statement.

In the presence of loops or recursion, the program semantics is described using the least fixed point.

3.2 Overview of Combined Heap and Value Analysis

The technical report [5] formalizes the combined heap and value analysis implemented in the static analyzer Sample. This section gives an informal overview of the analysis and illustrates it on examples.

Running Example. Consider the method `firstNaturals` in Listing 4. Given a positive integer n, the method builds a linked list of the natural numbers from 1 to n. The values increase by one when traversing the list and the last element of the list (whose `next` field is **`null`**) has the value n. The method builds the list starting with the last element.

3.2.1 Abstract Domain

The analysis abstracts concrete heaps as directed graphs, in which nodes represent concrete objects and edges represent possible references. Furthermore, each edge is associ-

```
var next: Ref
var val: Int

method firstNaturals(n: Int) returns (first: Ref) requires n > 0 {
  var tmp: Ref
  var i: Int
  first := null
  tmp := null
  i := n
  while (i > 0) {
    tmp := first
    first := new()
    first.val := i
    first.next := tmp
    tmp := null
    i := i - 1
  }
}
```

Listing 4: SIL method that builds a linked list of the first n natural numbers.

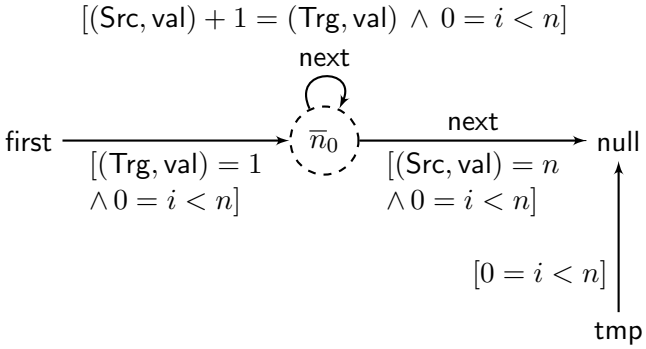


Figure 3: Abstract heap at the end of the method `firstNaturals`.

ated with a value state that approximates values of local variables and value fields of objects in which the edge may exist.

Figure 3 depicts the abstract state inferred by the analysis at the end of the method `firstNaturals`.

Nodes. The analysis distinguishes between *summary* and *definite* nodes. A summary nodes represents possibly many concrete objects, while a definite node represents a single concrete object. We denote summary nodes with dashed circles and definite nodes with solid circles. null is a special definite node. The formalization in [5] represents local variables as fields of a special definite node $\Im$.

In Figure 3, $\overline{n}_0$ is a summary node as it represents all elements in the list.

Edges. Edges in the abstract heap indicate possible targets of reference variables and reference object fields.

In Figure 3, the edge from first to $\overline{n}_0$ indicates that this local variable refers to an object represented by $\overline{n}_0$. Edges from $\overline{n}_0$ tell us that an object represented by $\overline{n}_0$ can point via next to either null or an object represented by $\overline{n}_0$.

Abstract Value States on Edges. Each edge in an abstract heap is associated with an *abstract value state* (also called *abstract condition*) that tracks the numerical values of local variables and fields. The analysis is parametrized by a value domain. In our examples, the analysis is parametrized by Polyhedra [2] implemented in Apron [8].

In Figure 3, the constraint $0 = i < n$ is present in all abstract conditions and means that this constraint holds in all possible states at the end of the method.

Edge-Local Identifiers. Abstract value states may contain *edge-local identifiers* that refer to value fields of the source or target node of the edge. Edge-local identifiers are useful to express invariants for edges whose source or target is a summary node.

In Figure 3, the edge-local identifier $(\mathsf{Trg}, \mathsf{val})$ in the value state on the edge from first to $\overline{n}_0$ refers to the value of the concrete object that the edge points to. The identifier does not refer to the value field of *all* objects represented by $\overline{n}_0$, only the one that first points to. Hence, the constraint $(\mathsf{Trg}, \mathsf{val}) = 0$ expresses that the first list element has the value 0.

The edge from $\overline{n}_0$ to null is associated with the constraint $(\mathsf{Src}, \mathsf{next}) = n$. This constraint expresses that an object represented by $\overline{n}_0$ whose next field points to null has the value n. That is, the value of the last list element is n.

Finally, the constraint $(\mathsf{Src}, \mathsf{next}) + 1 = (\mathsf{Trg}, \mathsf{next})$ on the self-loop of $\overline{n}_0$ mandates that the value of the next node is smaller by one then the predecessor node.

Heap Value Identifiers. In addition to edge-local identifiers, abstract conditions also contain *heap value identifiers* of the form $(\overline{n}, f)$ for a heap node $\overline{n}$ and value field f. Such an identifier refers to the field f of all objects represented by $\overline{n}$, which may be a summary node.

For the sake of readability, we omit constraints containing heap value identifiers in Figure 3. Edge conditions contain the constraint $1 \leq (\overline{n}_0, \mathsf{val}) \leq n$, indicating that the values of all list elements are between 1 and n.

3.2.2 Join

The join of two abstract heaps $\overline{\sigma}_1$ and $\overline{\sigma}_2$ is an abstract heap $\overline{\sigma}$ that overapproximates all concrete heaps that $\overline{\sigma}_1$ and $\overline{\sigma}_2$ represent.

The join minimizes the size of the resulting abstract heap via computing the graph representation of the join as a minimum common supergraph of σ_1 and σ_2. Smaller abstract heaps are easier to interpret and result into more efficient analysis.

Finding a minimum common subgraph of two graphs can be reduced into a problem of finding a maximum common subgraphs (MCS) of two graphs. In the resulting heap, the abstract conditions on the edges from the MCS are joins of abstract conditions on edges from the the source abstract heaps which formed the MCS.

Example. Figure 4 illustrates the join of two abstract heaps in the `firstNaturals` method. In the abstract state before entering the loop, both first and tmp point to null and the the edge conditions express that $i = n$. In the abstract heap at the end of the first loop iteration, first points to a newly allocated node $\overline{n}_0$ with value n, while tmp points to null and $i = n - 1$ is part of all edge conditions.

The conditions of the edge from $\overline{n}_0$ is the same as in the lower left-hand side heap as it is not in the maximum common subgraph of the left-hand side heaps. However, the condition on the edge from tmp to null is a join of the corresponding edges from the left-hand side heaps, as it participates in the maximum common subgraph.

3.2.3 Widening

The above join operator does not guarantee the convergence of the analysis as after every application of join, the resulting abstract heap may grow. The widening operator ensures the termination of the analysis by bounding the size of an abstract heap. The opeator achieves this by merging nodes into summary nodes. The analysis is parametric in the way now nodes are merged.

Example. Figure 5 illustrates a possible merge of nodes into a summary node in an abstract state of the `firstNaturals` method. Here, nodes $\overline{n}_0$, $\overline{n}_1$ and $\overline{n}_2$ are as a part of widening all merged into a single summary node $\overline{n}_0$.

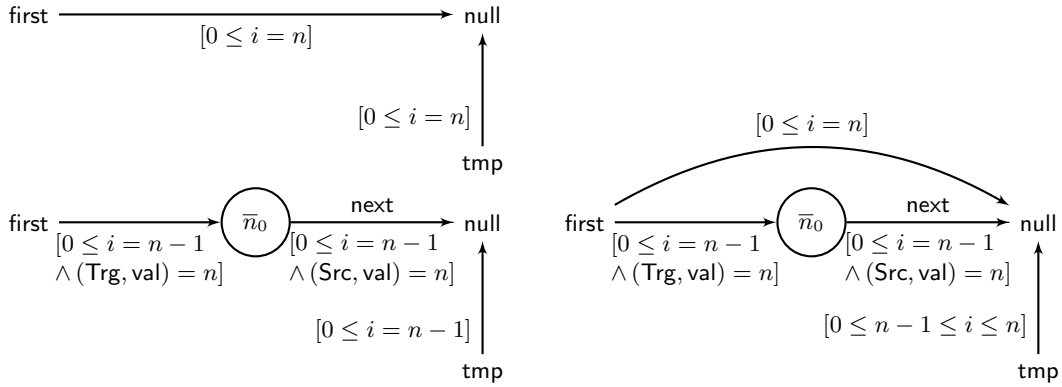


Figure 4: Abstract states before and after the first loop iteration of `firstNaturals` (on the left) and the join of them (on the right).

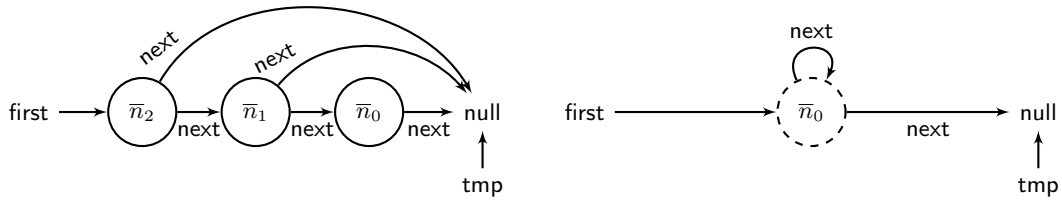


Figure 5: Abstract state before and after merging nodes at the end of the loop in `firstNaturals` in the third analysis iteration. Edge conditions have been omitted.

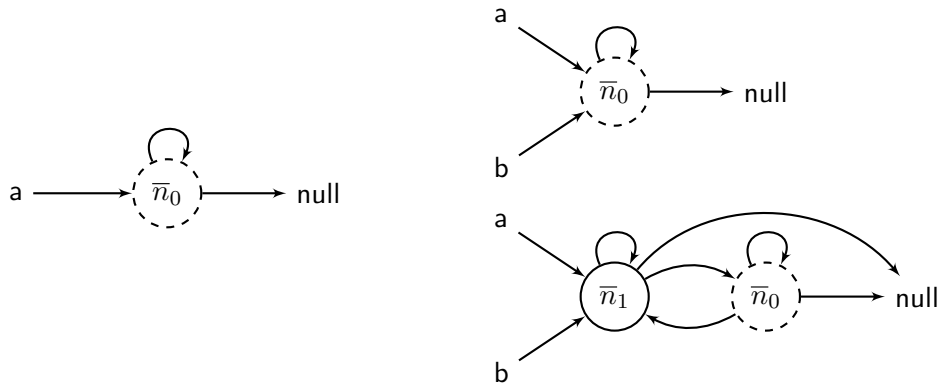


Figure 6: Assigning a to b in the left abstract heap yields the bottom-right abstract heap if materialization is enabled and the top-right abstract heap otherwise.

3.2.4 Materialization

The combined heap and value analysis offers a feature called *materialization* that can make the analysis more precise in the presence of summary nodes.

Whenever the program may dereference an object that is represented by a summary node, materialization creates a new separate definite node that represents that object. The analysis ensures that the new definite node also has all possible in-coming and out-going edges, based on the edges of the summary node. Materialization itself does not change the set of concrete heaps that the abstract heap represents.

However, in contrast to assignments to fields of summary nodes, assignments to fields of definite nodes allow for more strong updates, resulting in a more precise analysis. The main drawback of materialization is the blow up of the heap size, which hampers the efficiency of the analysis.

For this project, materialization is always enabled.

Example. Consider the abstract heap in Figure 6 on the left. Variable a refers to an object represented by a summary node $\overline{n}_0$. Suppose that we assign a to variable b.

Without materialization, b would simply point to $\overline{n}_0$ like a in the resulting abstract heap. This abstract heap does not imply that a and b are aliases. According to the abstract heap, they may refer to different objects.

With materialization enabled, the access to the variable a adds a new definite node $\overline{n}_1$ to the abstract heap that only represents the object a refers to. After the assignment of a to b, b will only point to $\overline{n}_1$. Since $\overline{n}_1$ only represents a single object, the abstract heap expresses that a and b are certainly aliases.

3.2.5 Edge Disambiguation Ghost State

Abstract heaps may contain multiple edges with the same source node and labeled with the same field, allowing the abstract heap to represent disjunctive information.

Suppose that a definite node $\overline{n}$ (representing only a single object) has multiple out-going edges labeled with the same field f. In any concrete heap, the field f of the represented object can only be a reference represented by exactly one of these edges of $\overline{n}$. That is, it is not possible for more than one of these edges to be present in any concrete heap. We say that these edges of $\overline{n}$ are *ambiguous*.

To decide whether some references in an abstract state are aliases, it may be necessary to know whether certain combinations of ambiguous edges are possible. The analysis itself does not explicitly track aliasing information.

In the following, we present how to leverage the existing heap analysis such that it maintains information on impossible combinations of edges in the abstract heap. As a result, the analysis provides more precise information about aliasing. This extension has been devised and implemented as a part of this project.

Example. Consider the SIL identity method shown in Listing 5. The method returns the reference `a` passed as a parameter by assigning it to the local result variable `b`. Parameter `a` could either be **`null`** or refer to some object.

Figure 7 shows the resulting abstract heap of the unmodified heap analysis on the left. Before the assignment, `a` has out-going edges pointing to both a definite node $\overline{n}_0$ and `null`. After the assignment, the local variable `b` also has edges to $\overline{n}_0$ and `null`. The abstract heap does not express that `a` and `b` are aliases.

Our solution is to introduce a local ghost variable i_0 at the beginning of the method that has a distinct value on all ambiguous edges of `a`. Concretely, i_0 is 0 if `a` is `null` and 1 if it points to $\overline{n}_0$. The local ghost variable is not part of the program, it only exists in the abstract state.

In general, the analysis introduces a ghost variable for each reference parameter of a method, and also when new ambiguous edges appear in abstract heaps due to materialization.

The right abstract heap in Figure 7 shows the result after assigning `a` to `b`. Both edges pointing to `null` have $i_0 = 0$ in their condition, while i_0 has the value 1 on the edges to $\overline{n}_0$. The abstract heap expresses that `a` and `b` are aliases. In a concrete heap, i_0 cannot have the value 0 and 1 at the same time. That is, such combinations of edges are impossible.

```
method id(a: Ref)
  returns (b: Ref)
{
  b := a
}
```

Listing 5: SIL identity method.

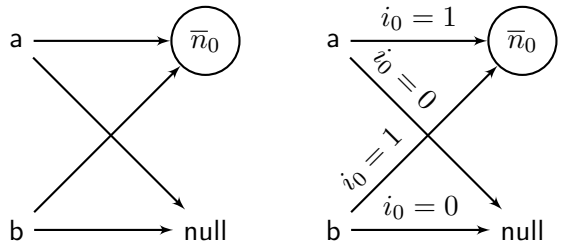


Figure 7: Abstract heap without and with ghost state after the assignment `b := a` in Listing 5.

Semantics of Ghost Variables. The analysis approximates the value of ghost variables in edge conditions in the same way as it approximates the values of other numerical variables and fields.

On many edges of the abstract heap, a ghost variable such as i_0 may have an unknown value ($\top$), meaning that the edge may exist no matter what the value of i_0 is. However, when the abstract states of i_0 in the conditions of two edges do not intersect (e.g., is 0 in one condition and 1 in the other), then it is guaranteed that these edges cannot both exist in a concrete heap.

4 Translation of SIL Programs

The static analyzer Sample uses a small, but expressive object-oriented input language called Simple [4]. In order to analyze a SIL program with Sample, it needs to be translated to Simple first.

Statements and Expressions. Simple does not distinguish between statements and expressions, a practice adopted by some programming languages such as Scala. Conceptually, statements such as assignments return an "empty" value. Furthermore, arithmetic (and other) operators are not part of the language definition. Instead, they are represented as method calls. For example, the expression `1 + 2` is represented as a method call `1.+(2)`. A part of the translation is to define the native semantics of such methods with respect to the abstract interpretation.

Simple does not accomodate control flow statements such as loops and conditionals. Instead, the body of each Simple method is a control flow graph. That is, we directly translate the CFGs of SIL methods rather than their ASTs.

For the supported data types **`Ref`**, **`Int`** and **`Bool`**, the translation preserves most basic expressions such as literals, unary and binray arithmetic and boolean expressions, conditional expressions as well as variable and field accesses.

Pre-Existing Specifications. The original SIL program may already contain partial specifications. To make the analysis more precise, it is desirable to preserve such specifications in the translation.

The translation preserves logical assertions such as `n > 0` in method preconditions and loop invariants. However, the translation does not support assertions that involve permissions, and ghost statements such as **`fold`** and **`unfold`**.

5 Running Examples and Associated Challenges

SIL methods that generate and traverse a linked list are simple examples of methods that operate on recursive data structures. In this section, we present these examples and show that analyzing them with the heap analysis described in Section 3 does not provide enough information to infer specifications for them. We will use the examples to illustrate our solution that can be generalized to other data structures such as trees.

5.1 List Generation

Listing 6 contains a SIL method `firstNaturals` with specifications, that takes as argument a positive integer n and returns a linked list whose elements have the values 1 to n in ascending order. We already used it to illustrate the heap analysis in Section 3.

The `firstNaturals` method would verify without any specifications. The only field accesses occur on a newly allocated object. Without any specifications, the method would not return permissions to any field of the generated list to its caller. The caller could not access any field of the generated list. Thus, the postcondition should contain the permission to the generated list, expressed with a recursive predicate instance.

Analyzing this method with the heap analysis yields the abstract heap shown in Figure 8, which was already discussed in Section 3.2.1. This abstract heap is not suitable for extracting the desired postcondition for the method. To extract a predicate instance recursing over `next` for `first`, we need to know that the abstract heap represents an acyclic linked list. Acyclicity follows from the conditions on the edges, but this information is not available explicitly.

5.2 List Traversal

Listing 7 contains a SIL method with specifications, that takes a linked list as argument. The body of the method traverses every element of the list and sets the value of each element to zero. In this program, the local variable `cur` always points to the current element of the list. The termination criterion of the loop is that `cur` reaches **`null`**.

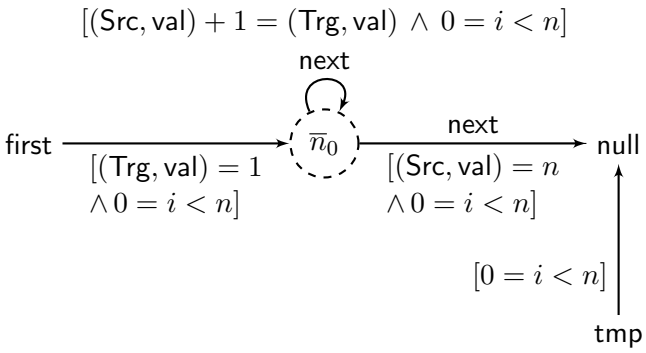


Figure 8: Abstract heap at the end of the method `firstNaturals`.

```
var next: Ref
var val: Int

method firstNaturals(n: Int) returns (first: Ref)
  requires n > 0
  ensures acc(valid(first))
{
  var tmp: Ref
  var i: Int
  tmp := null
  first := null
  i := n
  while (i >= 0)
    invariant (first != null) ==> acc(valid(first))
  {
    tmp := first
    first := new()
    first.val := i
    first.next := tmp
    tmp := null
    i := i - 1
    fold acc(valid(first))
  }
}

predicate valid(this: Ref)
{
  acc(this.next) && acc(this.val) &&
  (this.next != null ==> acc(valid(this.next)))
}
```

Listing 6: Specified SIL method that builds a list of the first n natural numbers.

```
var val: Int
var next: Ref

method setListToZero(list: Ref)
    requires (list != null) ==> acc(valid(list))
{
    var cur: Ref
    cur := list
    while (cur != null)
        invariant (cur != null) ==> acc(valid(cur))
    {
        unfold acc(valid(cur))
        cur.val := 0
        cur := cur.next
    }
}

predicate valid(this: Ref)
{
    acc(this.val, write) && acc(this.next, write) &&
    ((this.next != null) ==> acc(valid(this.next), write))
}
```

Listing 7: SIL method with specifications, that sets all elements of a linked list to 0.

Unlike the list generation example, Listing 7 would not verify without specifications. Rather than building the data structure itself, the method operates on a data structure that the caller must supply. Thus, the precondition must contain the appropriate permissions, expressed with a recursive predicate instance.

To infer the loop invariant for `setListToZero`, we need to know that before and after every loop iteration, the local variable `cur` is either **`null`** or points to an acyclic linked list, recursing over `next` and terminated with **`null`**. As we will show next, the result of the unmodified heap analysis does not provide this information.

Initial Abstract Heap in Unmodified Heap Analysis. The unmodified heap analysis analyzes the method starting from the abstract heap shown in Figure 9. Since there is no precondition, the heap analysis does not assume anything about the heap in the initial state. The summary node $\overline{n}_0$ represents any set of objects whose next field either points to null or some object (possibly themselves).

Resulting Abstract Heap in Unmodified Heap Analysis. Figure 10 shows the resulting fixed point abstract state at the guard of the loop, which we would like to extract the loop invariant from.

The definite node $\overline{n}_1$ in the abstract heap represents the object that list and cur pointed to

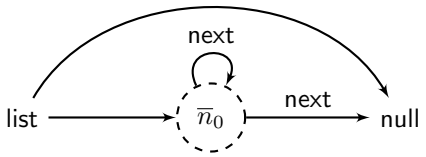


Figure 9: Initial abstract heap at the beginning of the method `setListToZero`.

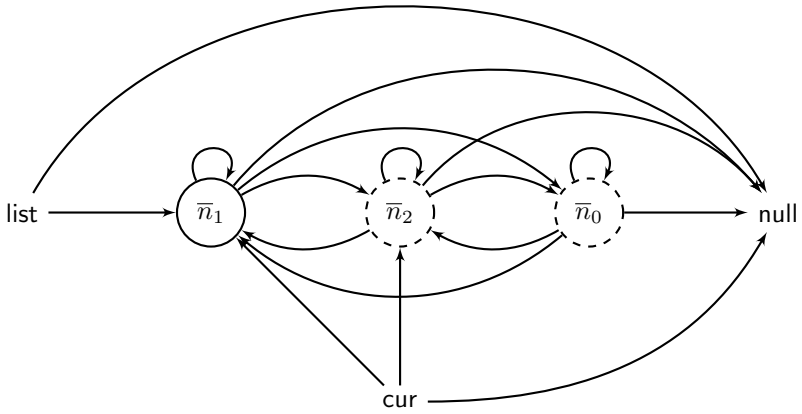


Figure 10: Abstract fixpoint heap at the loop guard of the method `setListToZero`. Labels of next fields omitted for readability.

before entering the loop. The summary node $\overline{n}_2$ represents all objects that cur could have pointed to in any of the past loop iterations (except the one before entering the loop). Together, $\overline{n}_1$ and $\overline{n}_2$ represent the part of the heap that the loop has "explored", while the summary node $\overline{n}_0$ represents all objects in the "unexplored" part of the heap.

This abstract heap is not suitable for extracting the loop invariant, for several reasons.

Disjunctive Information The abstract heap contains many nodes that have multiple out-going edges, that represent different possible shapes of the heap, and the extracted loop invariant should hold for all of them. Thus, in the loop invariant, it may be necessary to distinguish between different shapes.

For the variable list, the distinction between its null and non-null target is easy. However, it would in general not be possible to distinguish in the loop invariant between the two possible non-null targets of cur. The disjunctiveness of the abstract heap is even worse for the definite and summary nodes. Each of these nodes points to every other node.

In the loop invariant, there should be a predicate instance with argument cur that specifies the remainder of the list. However, this importance of the variable cur over list is not clear from the abstract heap.

Cycles The abstract heap also represents cyclic data structures that could not be spec-

ified with a recursive predicate.

Unrelated Objects The summary nodes in the abstract heap could represent other objects that do not belong to the list that the method recurses over. These objects are not described by the predicate that specifies the list.

The next section will present our analysis and explain how it addresses the above problems by tracking additional information in abstract heaps.

6 Analysis for Inferring SIL Predicates and Predicate Instances

The previous section has illustrated that the results of the heap analysis do not provide sufficient information to infer specifications for methods that operate on recursive data structures. In this section, we will present how to extend the heap analysis such that analyzing a method yields sufficient information to extract predicates and predicate instances for that method's specification.

Section 6.1 introduces the class of predicates that our analysis aims to infer. Section 6.2 and Section 6.3 explain what information our analysis tracks with respect to predicate definitions and predicate instances, respectively. Later, Section 6.4 and Section 6.5 introduce how this information is accumulated during the analysis, specifically how the analysis infers recursive predicates. Then, Section 6.6 presents how the heap analysis can be made more precise after the analysis has inferred that a predicate is recursive. Later sections discuss orthogonal topics that allow the analysis to infer specifications for a wider range of programs.

6.1 Supported Predicates

We have designed the analysis to be able to handle programs whose specification can be expressed with a restricted class P of predicates. These predicates are still expressive enough for the specification of non-trivial methods that operate on recursive data structures.

In the following, we describe this class of predicates. In later sections, we will make clear how the analysis uses these restrictions.

Assumption 6.1. *A predicate p in class P has exactly one reference parameter.*

In this section, we usually call the parameter `this`, as it can be thought of as the *receiver* of the predicate.

Since predicate instances are always acyclic (see Section 2.3), it is impossible get an instance of a predicate in class P with the same argument by unfolding. This implies that the data structure described by such a predicate instance is acyclic with respect to the field of recursion.

Assumption 6.2. *The body of predicate p in class P is a conjunction of*

- *field access predicates* `acc(this.`f`)` *for fields f, and*
- *conditional predicate instances* `(this.`f_r `!= null) ==> acc(`q`(this.`f_r`))` *for reference fields f_r and a predicate q in class P.*

Field Access Predicates. According to Assumption 6.2, field access predicates are unconditional. That is, an expression such as `(this != null) ==> acc(this.`f`)` cannot occur in the body of a predicate. This restriction implies that the predicate argument `this` for every instance `acc(`p`(this))` must not be `null`.

Nested Predicate Instances. The access predicate `acc(`q`(this.`f_r`))` can only occur in the predicate if the predicate also contains permission to the field f of `this`. Otherwise, the access predicate would not be framed (see Section 2.2).

If the nested predicate p is the same as p, then p is directly recursive. However, q could also be a different predicate than p.

No Logical Assertions. Logical assertions are assertions that do not involve permissions, such as relationships between the values of fields. In general, SIL predicate bodies can contain such logical assertions, as long as there are self-framed (see Section 2.2). For example, the predicate for the method Listing 6 could also contain the assertion that the linked list is sorted.

For simplicity, we do not support logical assertions in the body of predicates.

6.2 Tracking Predicates

Our analysis incrementally builds the predicates for the specification of a method, based on how the method operates on the heap. When analyzing a method, we do not decide up-front what the predicates for the method should be.

In the following, we will explain how we represent predicates in the analysis, and how the analysis incrementally builds these predicates.

6.2.1 Representation of Predicates

In our analysis, each abstract state contains a representation of one or more predicates described in Section 6.1. Such a predicate can be represented unambiguously with the following information:

- The unique name of the predicate, referred to as a *predicate identifier*.
- The set of fields that the predicate contains the permission for.

  Since we assume that the predicate has a single reference parameter (Assumption 6.1), the predicate only contains permissions for fields of the parameter object. The name of the parameter in a predicate can be chosen arbitrarily. Thus, the representation of the predicate does not need to include it.
- For each reference field with permission, the set of predicates the predicate contains a nested instance of.

Thus, it is possible to compactly represent such a predicate only with a predicate identifier and an associated predicate body that stores the fields with permission and any nested predicate instances. In this report, we use symbols such as $\overline{p}$, $\overline{p}_0$ and $\overline{p}_1$ to identify predicates.

Notation. Consider the SIL predicate in Listing 8 that describes a linked list. For the representation in the analysis, the only relevant information is that the predicate contains permission for the field `val` and recurses over `next`. Assuming that we use use the predicate identifier $\overline{p}$, we denote such a linked list predicate as $\overline{p} \mapsto [\mathsf{val}, \mathsf{next} \rightarrow \{\overline{p}\}]$.

```
predicate p(list: Ref) {
  acc(list.next) && acc(list.val) &&
  (list.next != null ==> acc(p(list.next)))
}
```

Listing 8: Recursive predicate for a linked list.

6.2.2 Predicates as a Mechanism for Tracking Permissions

The method may operate on a data structure whose specification does not necessarily need a predicate. For example, when the method operates on a non-recursive data structure such as a simple object with a field, then simple field access predicates (see Section 2.2) are sufficient for the specification.

The analysis uses predicates to track permissions for all data structures, even non-recursive ones. That is, the analysis may infer that the predicate $\overline{p} \rightarrow [\mathsf{val}]$ with a single permission for field val can be used in the method's specification. After the analysis, however, it is not necessary to actually use the inferred predicate and instances of the predicate in the specification. Instead, the field permissions in the inferred predicate can also be added to the specification directly.

In summary, the analysis uses predicates as a mechanism to track *all* permissions.

Adding Permissions to Predicates. The analysis gradually populates predicates with permissions and nested predicates. Since predicates track all inferred information about permissions, the analysis tries to identify for each field access in the method the predicate that the corresponding permissions should be a part of.

The analysis never removes permissions from predicates. That is, in the state at the end of the analysis, the predicates contain all permissions that were added at any point of the analysis.

In Section 6.4, we will explain in more detail how permissions are added to predicates.

6.2.3 Predicates as Placeholders

Since the analysis gradually adds permissions to predicates, predicates are initially empty. That is, the analysis has not yet associated and permissions with such a predicate. We denote a predicate $\overline{p}$ that is empty as $\overline{p} \mapsto \top$.

An empty predicate is essentially a placeholder for a predicate. If the analysis never adds any permissions to a predicate, then the predicate is not needed and can be discarded

at the end of the analysis. Recall that it is not necessary to extract an actual SIL predicate from an inferred predicate in the analysis result. For empty predicates, this holds true especially, because a predicate with an empty body is not useful in the inferred specification.

The analysis may introduce many such placeholder predicates, even though they may not be necessary in the end.

Example. Suppose that the analysis adds permission for a reference field f_r to an empty predicate $\overline{p}_0 \mapsto \top$. At this point, the analysis does not yet know whether predicate $\overline{p}_0$ should also contain some nested predicate instance for field f_r. Thus, the analysis creates a new nested predicate that acts as a placeholder. The new predicate has an identifier such as $\overline{p}_1$ that has never been used before (called *fresh*) and its body is $\top$.

As a result, there are now two predicates $\overline{p}_0 \mapsto [f_r \to \{\overline{p}_1\}]$ and $\overline{p}_1 \mapsto \top$. $\overline{p}_0$ contains a nested instance of predicate $\overline{p}_1$ for field f_r.

The new predicate state expresses that $\overline{p}_0$ could potentially have a nested predicate instance. This gives the analysis the ability to add permissions if it needs to.

6.2.4 Detecting Recursive Predicates By Merging Predicates

Two predicates in the analysis do not necessarily need to appear as separate predicates in the inferred specification. Instead, the analysis can decide that two predicates should be *merged*, such that the resulting predicate contains all the permissions of both predicates.

Merging predicates is the mechanism that allows the analysis to implicitly detect that a predicate should be recursive. Section 6.5 will explain and illustrate the mechanism in more detail.

6.2.5 Restricting Heap Analysis for Known Recursive Predicates

Suppose that the analysis has already inferred that the method operates on a data structure that is described by a recursive predicate. Such a recursive predicate guarantees that the data structure is acyclic with respect to the field of recursion. The predicate imposes constraints on the possible shapes of the heap.

We can exploit these constraints to make the heap analysis more precise by removing edges from the abstract heap that are impossible for an acyclic data structure.

Section 6.6 will present the involved challenges and how we address them.

Example. The left abstract heap in Figure 11 contains a local variale list that points to a summary node $\overline{n}_0$. Recall that accessing variable list triggers a materialization (see Section 3.2.4) of a definite node $\overline{n}_1$ that list points to.

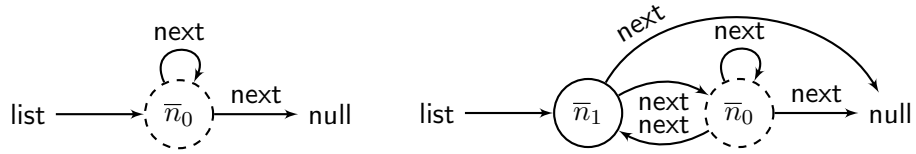


Figure 11: Abstract heaps before and after the materialization of the target node of a variable list.

Suppose that the analysis has already inferred that there is a predicate instance recursing over next at the object represented by $\overline{n}_1$. If the field next of that object pointed to the object itself, this would mean that the predicate instance contains an nested predicate instance with the same receiver. Since such a cyclic instance cannot exist, so the analysis can safely remove such the self-loop at $\overline{n}_2$ labeled with next.

6.3 Tracking Predicate Instances

In addition to predicate, our analysis also tracks where instances of these predicates are in the abstract heap for each point in the method. The information on predicate instances tells the analysis which predicates to add permissions to.

6.3.1 Representation of Predicate Instances

The predicates supported by our analysis only have a single reference parameter, by Assumption 6.1. Thus, an instance can be represented by the identifier of the predicate and its receiver object. The restriction to a single parameter is very useful for representing a predicate instance in the abstract state, as we will show now.

We use extra *ghost* fields of an object to represent holding a predicate instances at that particular object. The heap analysis offers a mechanism for the abstract condition on an edge to refer to fields of the source and target objects of the edge: *edge-local identifiers* (see Section 3.2.1) do exactly that.

By using an edge-local identifier $(\mathsf{Trg}, \overline{p})$ in the abstract condition on some edge, we can identify an instance of predicate $\overline{p}$ whose receiver is the target object of the edge. That is, the target object has a ghost field named $\overline{p}$.

The analysis never uses source edge-local identifiers $(\mathsf{Src}, \overline{p})$ to identify predicate instances whose argument is the source object of the corresponding edge. Predicate instance identifiers *always* refer to the target object of the edge. Thus, we can make Trg implicit for identifying predicate instances.

6.3.2 Folded and Unfolded Labels

To indicate that there is a folded instance of a predicate $\overline{p}$ at some object, the analysis assigns the value Folded to the ghost field $\overline{p}$ of that object.

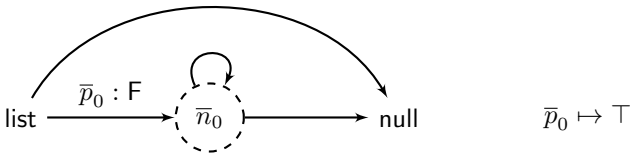


Figure 12: Initial abstract state for the `setListToZero` method.

When the method accesses a field of an object for which there is a folded instance of $\overline{p}$, then the analysis infers that the permission for that field should be part of $\overline{p}$. That is, the information on predicate instances guides the analysis in populating predicates with permissions.

In the specification of a SIL program, it is necessary to unfold a predicate instance that contains the permission to a field before accessing that field (see Section 2.2). After unfolding, the method does not hold the predicate instance any more, only the permissions stored in it.

The analysis reflects this by replacing the value Folded of the object's ghost field $\overline{p}$ with the value Unfolded, The Unfolded value allows the analysis to remember that there used to be a folded predicate instance of $\overline{p}$ at that object. As we will explain in Section 6.6, this information is central for making the heap analysis more precise.

There is another advantage to distinguish between folded and unfolded instances: replacing Folded with Unfolded at some point in the method suggests that the method's specification should contain a corresponding **`unfold`** statement at that point.

Example. Consider the abstract heap graph shown in Figure 12. The edge from list to the summary node $\overline{n}_0$ contains the label $\overline{p}_0$: F. F is shorthand for Folded. The label on this edge indicates that the ghost field $\overline{p}$ of the target object of the edge has the value Folded. Thus, when list does not point to null, the method holds a folded instance of predicate $\overline{p}$ for the object that list points to.

The right-hand side of Figure 12 shows that $\overline{p}$ is just a placeholder predicate in this abstract state.

6.3.3 Initial Abstract State of a Method

For the initial abstract state of a method, the analysis creates a fresh placeholder predicate $\overline{p}_r$ for every reference parameter r.

The analysis assumes that the method holds a folded instance of $\overline{p}_r$ with argument r in the case that r is not null. Hence, it adds a label $\overline{p}_r$: Folded to the edge of r that does not point to null. The reason why the null edge does not have a label is our Assumption 6.2 about the shape of $\overline{p}_r$, which does not allow null as argument.

The analysis cannot infer multiple distinct predicates for a single parameter r. The reason is that it would not be clear what predicate to add permissions to when the

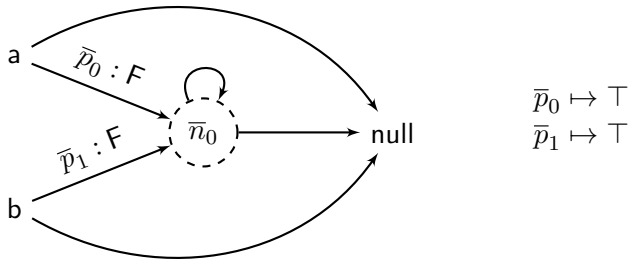


Figure 13: Abstract initial state of a method with two reference parameters a and b.

method accesses any fields of parameter r.

Examples. The abstract state in Figure 12 shown earlier is exactly the initial state that our analysis uses for the `setListToZero` method. The analysis assumes that if list is not null, then the method holds an instance of $\overline{p}_0$ for list. The analysis does not associate any permissions with $\overline{p}_0$ yet.

Figure 13 shows a more general initial abstract state of a method with two reference parameters a and b. The analysis creates a fresh predicate for each of them.

6.3.4 Variable Accesses and Assignments

Local variable accesses of a method do not require any modifications to the predicate-related abstract state. In SIL, accessing local variables does not require permissions, because they only exist on the call stack.

Accessing a local reference variable with an edge to a summary node in the abstract heap triggers a materialization of the object that the variable points to (see Section 3.2.4).

For both materialization and local variable accesses, the semantics of our analysis does not differ from the semantics of the unmodified heap analysis. The heap analysis treats predicate instance identifiers like all edge-local identifiers.

Example. Figure 14 contains the abstract heap after the assignment `cur := list` in `setListToZero`. Accessing the variable list has caused the heap analysis to materialize the node $\overline{n}_1$. list and cur point to the same definite node $\overline{n}_1$. Both of them have $\overline{p}_0 : \mathsf{F}$ on their edge to $\overline{n}_1$.

The abstract heap in Figure 14 expresses that the method holds a *single* folded instance of $\overline{p}_0$ whose argument is the object represented by target node $\overline{n}_1$. That object can be referred to as list and cur, since they are aliases.

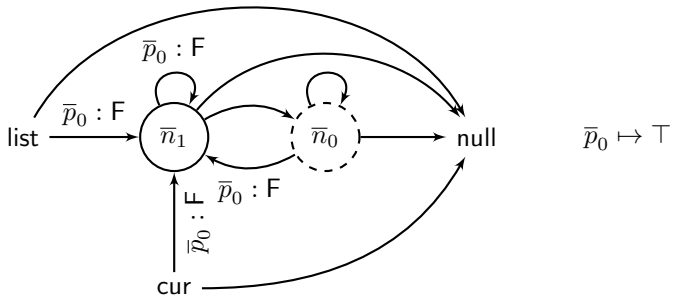


Figure 14: Abstract heap after the assignment `cur := list` in `setListToZero`.

6.4 Adding Permissions to Predicates

We have already mentioned earlier that the analysis adds permissions to predicates at points where the method accesses fields. Next, we will take a closer look at this operation and illustrate it with an example.

Constraint on Field Accesses. This report only discusses field accesses of the form `a.b`, that is, field accesses on local variables. Such field accesses are easier to handle as they do not require a recursive unfolding and adding permissions to multiple fields to different predicates. SIL programs can be rewritten to avoid deep field accesses by using temporary local variables. Thus, this limitation does not fundamentally limit the class of programs that our analysis can handle.

6.4.1 Identifying the Predicate Instance

Suppose that the method accesses a field f of a local variable o. The analysis then determines what predicate instance the permission for this field should come from.

The analysis only considers a predicate $\overline{p}$ for which there is a folded or unfolded instance at o. That is, the ghost field o.$\overline{p}$ must either be Folded or Unfolded. One of these predicates may already contain the permission for f, for example due to an earlier access to the same field. Otherwise, the analysis arbitrarily picks one of the predicates to add permissions to. In our examples, there is always just one predicate instance at o.

If the instance of the chosen predicate is still folded, then the analysis unfolds it first, by replacing the label Folded with Unfolded. Unfolding also causes nested predicate instances to appear in the abstract heap, if there are any. We will later explain in detail what happens if the instance being unfolded already contains nested predicate instances.

Example. Figure 15 depicts the abstract state before the field access `cur.val` in the first loop iteration of `setListToZero`. By entering the loop body, the analysis has

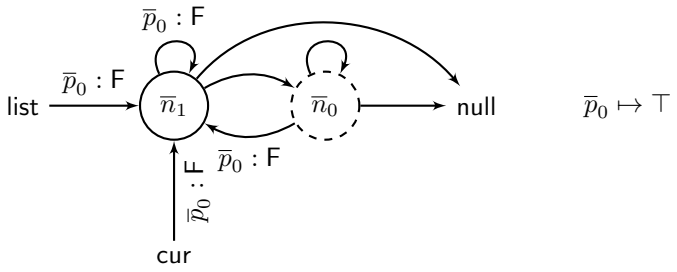


Figure 15: Abstract heap before the assignment `cur.val := 0` in the first loop iteration of `setListToZero`.

assumed that `cur` is not null. Thus, cur and list (an alias) do not point to null any more. The abstract heap contains only a single edge for cur and $\overline{p}_0$: F is the only label on that edge. Hence, the analysis concludes that the permission for the field val must be part of predicate $\overline{p}_0$.

6.4.2 Adding Permissions to Identified Predicate

The predicate $\overline{p}$ will contain the permission for f after the field access. After the assignment `cur.val := 0` in the `setListToZero`, predicate $\overline{p}$ then has the body [val] .

Reference Field Permissions. If the accessed field f is a reference field, then the predicate $\overline{p}$ may require a nested predicate instance for that field, for example if the method later accesses a field of the object that o.f points to. Thus, the analysis creates a fresh placeholder predicate $\overline{p}_{nested}$ and uses it as a nested predicate for field f in $\overline{p}$.

Thus, predicate $\overline{p}$ contains a new entry $f \to \{\overline{p}_{nested}\}$ after the field access and there is a fresh predicate $\overline{p}_{fresh} \mapsto \top$. The method now holds a folded instance of this placeholder predicate $\overline{p}_{fresh}$ with argument o.f, if o.f is not null. The analysis assigns the value Folded to the ghost field o.f.$\overline{p}_{fresh}$ to achieve this.

Example. Figure 16 depicts the abstract state of `setListToZero` in the first loop iteration after interpreting `cur.next`, but *before* assigning it to `cur`.

The field access `cur.val` has caused the analysis to unfold the instance of predicate $\overline{p}_0$ by assigning Unfolded to cur.$\overline{p}_0$#0. Because of this assignment, there is a label $\overline{p}_0$#0 : U on all edges into $\overline{n}_1$, since $\overline{n}_1$ represents the object that cur points to. Also, the analysis has added the permission to val to predicate $\overline{p}_0$.

Later, the field access `cur.next` has caused the materialization (see Section 3.2.4) of the definite node $\overline{n}_2$. The analysis has also added the permission next $\to \{\overline{p}_1\}$ to predicate $\overline{p}_0$ and a fresh placeholder predicate $\overline{p}_1$. In addition, the non-null edges going out of the node $\overline{n}_1$ now have the label $\overline{p}_1$: F. That is, the method holds an instance of the placeholder predicate $\overline{p}_1$ at cur.next, unless cur.next is null.

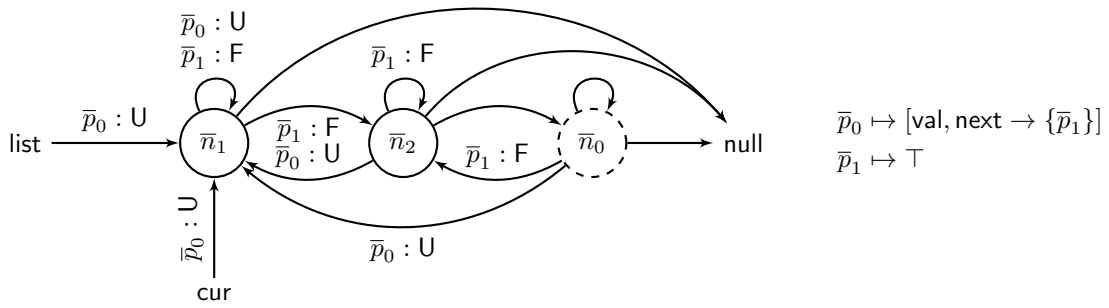


Figure 16: Abstract state after the field access `cur.next` in the first `setListToZero` loop iteration.

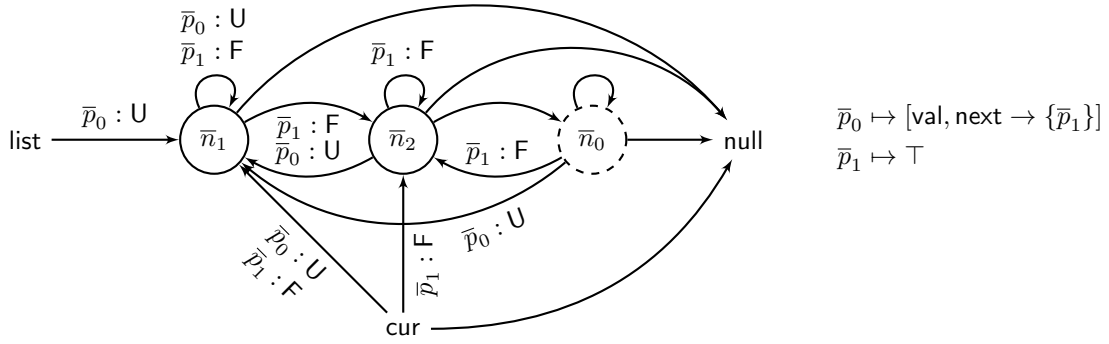


Figure 17: Abstract state at the end of the first `setListToZero` loop iteration.

Figure 17 depicts the abstract state of `setListToZero` in the first loop iteration *after* the assignment `cur = cur.next`. cur could now either point to $\overline{n}_1$, $\overline{n}_2$ or null, because these where the three possible targets of edges going out of $\overline{n}_1$ in Figure 16. If cur does not point to null, the method now certainly holds an instance of the placeholder predicate $\overline{p}_1$ at cur.

6.5 Detecting Recursive Predicates During Joining

So far, we have illustrated how the analysis adds permissions to the predicate that specifies the list parameter in the `setListToZero` example. Up until the end of the first loop iteration, the analysis has not yet assumed that the predicate is recursive. So far, all operations of the method were explainable with non-recursive predicates.

The heap analysis joins the abstract state at the end of the first loop iteration with the abstract state before entering the loop (Section 3.1). As we will show next, the analysis may merge predicates when joining two abstract states.

Deciding that two predicates should in fact be the same is the mechanism that allows the analysis to implicitly detect recursive predicates.

6.5.1 Merging Predicates

Suppose there is a local variable o for which the method holds folded instances of different predicates $\overline{p}_i$ and $\overline{p}_j$ in two abstract states $\overline{\sigma}_1$ and $\overline{\sigma}_2$ to be joined.

In such a situation, the analysis considers the predicates $\overline{p}_i$ and $\overline{p}_j$ to be the same from that point on. That is, the analysis *merges* the predicates $\overline{p}_i$ and $\overline{p}_j$ into just one predicate. The resulting predicate contains all permissions of both of them. By merging the predicates, the method can continue holding a predicate instance at o.

Target Predicate Identifier. In principle, the identifier of the resulting predicate could be either $\overline{p}_i$, $\overline{p}_j$, or even a fresh one. As the target identifier, our analysis chooses the one that was created earlier. The motivation is that the older predicate identifiers are used initially for reference parameters of the method and newer predicate identifiers often appear temporarily due to field accesses and object allocations.

Example. Consider the abstract states of `setListToZero` before entering the loop and at the end of the first loop iteration in Figure 14 and Figure 17, respectively. In the abstract heap before entering the loop, the method holds an instance of predicate $\overline{p}_0$ for cur. In the other abstract heap, it is an instance of predicate $\overline{p}_1$. Hence, the analysis merges $\overline{p}_0$ and $\overline{p}_1$ into $\overline{p}_0$ in both heaps.

In the abstract state before the loop, merging has no effect because the state does not contain the predicate $\overline{p}_1$ at all.

The abstract state after the loop contains the following predicates:

$$\overline{p}_0 \mapsto [\mathsf{val}, \mathsf{next} \to \{\overline{p}_1\}]$$
$$\overline{p}_1 \mapsto \top$$

In this state, all occurrences of $\overline{p}_1$ are replaced with $\overline{p}_0$. Furthermore, the predicate body of $\overline{p}_1$ is merged into the body of $\overline{p}_0$, which has no effect as $\overline{p}_1$ is empty. Thus, the resulting predicate state is

$$\overline{p}_0 \mapsto [\mathsf{val}, \mathsf{next} \to \{\overline{p}_0\}]$$

In the resulting state, predicate $\overline{p}_1$ does not occur any more. Instead, the merged predicate $\overline{p}_0$ is now recursive over next.

6.6 More Precise Heap Analysis for Recursive Predicates

Knowing that some predicate instance in the heap is recursive allows the analysis to exploit the constraint that the predicate instance imposes on the shape of the heap.

As explained in Section 6.1, predicate instances in our analysis always specify data structures that are acyclic with respect to the field of recursion. This mandates that if

there is a predicate instance recursing over a field f at some object o, then the object cannot have a self-loop for field f. Furthermore, the current instance at o may have been obtained by unfolding an instance of the same predicate at some other object q. Since predicate instances are acyclic, it is impossible to unfold the instance at o and get an instance at the object q. Thus, the object o cannot possibly point to object q via field f.

In this section, we will explain how the analysis can remove such impossible edges in a sound manner. By removing such impossible edges, we can reduce the amount of disjunction in abstract heaps, and as a result, it becomes more likely that the analysis yields an abstract heap that can be easily described using predicates.

6.6.1 Rerunning the Analysis

Suppose that the analysis infers that a predicate describing a method parameter is recursive. Our approach is not to try to retroactively make the abstract heap more precise. Instead, we rerun the analysis from an initial state that already contains the recursive predicate, such that the analysis can remove impossible edges from the beginning.

Such an approach is simpler and more robust. The rerun is expected to happen early, such as after the first loop iteration. Thus, rerunning should only have a minor impact on performance.

Next, we will show that the information on predicates and predicate instances tracked by the analysis is in general only enough to remove impossible self-loops. To remove impossible edges between nodes, we will need to track more information.

6.6.2 Impossible Self-Loops

If the abstract heap contains a recursive predicate instance for some object represented by a definite node, then the analysis can remove the self-loop for the recursion field from the node.

Figure 18 illustrates that the abstract state at the beginning of the loop in `setListToZero` does not have a self-loop any more on the node that represents the first list element.

Materialized Out-going Edges of Summary Nodes. Note that this abstract heap still contains an edge from the summary node $\overline{n}_0$ to the definite node $\overline{n}_1$. In general, it would not be sound to remove such an edge.

Suppose that there are other variables pointing into the summary node, for example method parameters (see Section 6.3.3). In such a case, the summary node could represent more objects than just the elements that belong to list. There could be objects that are not part of the list, and point to the first list element represented by $\overline{n}_1$. For example, the summary node may represent an object that is a predecessor of list. That is, such an edge could exist.

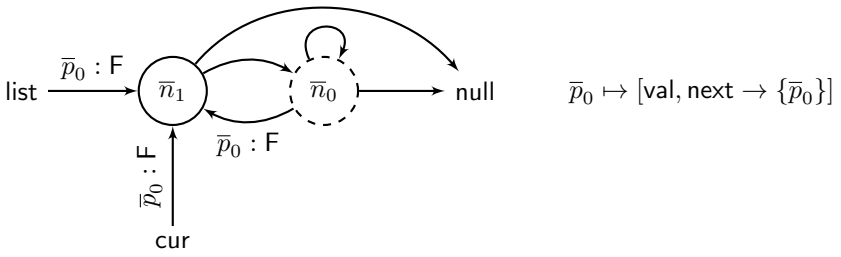


Figure 18: Abstract state at the beginning of the first loop iteration in `setListToZero`, during the rerun of the analysis. The self-loop of $\overline{n}_1$ was removed.

Effect on the Precision of the Analysis. Removing the self-loop from the first list element results in a more precise abstract heap at the end of the first loop iteration. As shown in Figure 19, cur only points to $\overline{n}_2$, representing the second list element, or null, if there is no such element.

Note that during the first loop iteration, the predicate instance $\overline{p}_0$ at $\overline{n}_1$ was unfolded. As the predicate already contained the permission for both field val and next, there was no need to add any permissions to $\overline{p}_0$ and create a placeholder predicate for next.

6.6.3 Impossible Edges to Already Unfolded Nodes

In Figure 19, there is an edge from node $\overline{n}_2$ to $\overline{n}_1$, that is, from the second to the first list element. The predicate $\overline{p}_0$ does not allow such an edge, because the current instance of $\overline{p}_0$ at $\overline{n}_2$ was unfolded from an instance of $\overline{p}_0$ at $\overline{n}_1$, and predicate instances cannot be cyclic.

The label $\overline{p}_0$: Unfolded for node $\overline{n}_1$ expresses that there used to be some folded instance of $\overline{p}_0$ at $\overline{n}_1$. However, this information is not enough to soundly remove the back-edge from $\overline{n}_2$ to $\overline{n}_1$. The problem is that the instance of $\overline{p}_0$ at $\overline{n}_2$ may not have been unfolded recursively from the instance at $\overline{n}_1$, but from some other instance of $\overline{p}_0$. In such a case, we could not exploit the acyclicity of the predicate instance.

In general, the analysis needs a way to be certain that some predicate instance was unfolded recursively from some other predicate instance. The following section illustrates how tracking additional information can solve the problem.

6.6.4 Version Numbers for Predicate Instances

We extend the representation of predicate instances in the analysis to express that certain predicate instances in the abstract heap were all unfolded from the same predicate instance. The additional information allows the analysis to remove impossible edges.

Concretely, a predicate instance at some object is now not only identified by a predicate $\overline{p}$, but also by a *version number* $\overline{v}$. That is, the edge-local identifier of a predicate

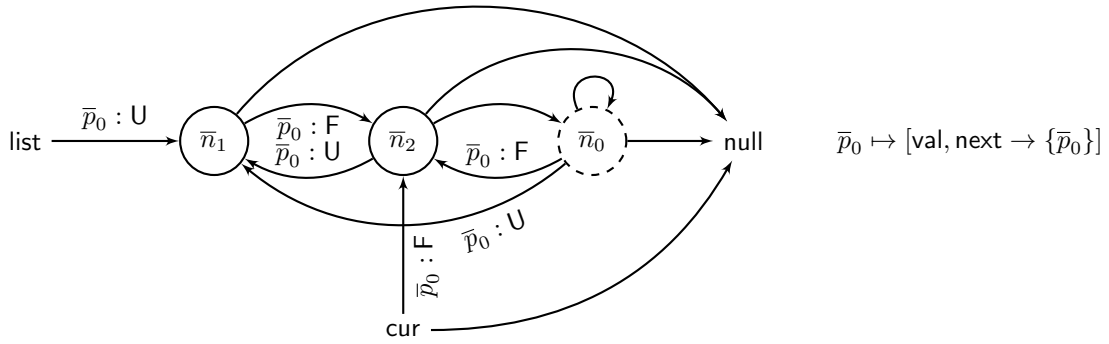


Figure 19: Abstract state at the end of the first loop iteration in `setListToZero`, during the rerun of the analysis. The self-loop of $\overline{n}_2$ was removed.

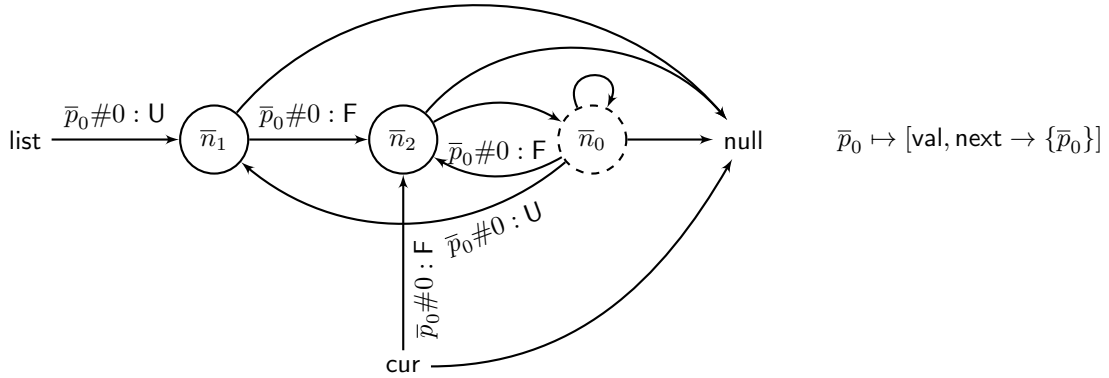


Figure 20: Abstract state at the end of the first loop iteration in `setListToZero`, with predicate instance version numbers.

instance is now a tuple of $\overline{p}$ and $\overline{v}$, which we denote as $\overline{p}\#\overline{v}$. The analysis uses integers as version numbers, starting from 0.

When unfolding a predicate instance with version number $\overline{v}$, all predicate instances that were contained in the instance have the same version number $\overline{v}$. This means that in an abstract heap, folded and unfolded instances with the same version number were all unfolded from the same predicate instance.

Example. Figure 20 again depicts the abstract state of `setListToZero` at the end of the first loop iteration. But in this abstract heap, the version number 0 was used for the instance of predicate $\overline{p}_0$ describing the parameter list.

After unfolding the predicate instance at $\overline{n}_1$, there is now a folded instance of $\overline{p}_0$ at $\overline{n}_2$ with the same version number 0, denoted with the label $\overline{p}_0\#0$: Folded. The version number 0 expresses that the predicate instance at $\overline{n}_2$, was contained in the predicate instance that used to be at $\overline{n}_1$. Since predicate instances cannot contain nested instances with the same receiver, it is sound to remove the edge from $\overline{n}_2$ to $\overline{n}_1$ for field next.

6.6.5 Joining Precise Abstract Heaps

In the previous section, we have shown how version numbers allow the analysis to remove impossible edges and make the abstract heaps more precise. Before we can discuss the result of the analysis rerun for our running example, we need to take a look at how the behavior of joins in the heap can change if the abstract heaps are more precise.

When the heap analysis joins abstract heaps, there may be more than one possible result, because the result depends on how the analysis matches nodes in the two abstract heaps (see Section 3.2.2). Even though all of these possible results are sound, not all of them are equally useful for specification inference, as we will explain next.

Figure 21 depicts the precise abstract heaps before and after the first loop iteration, which are to be joined. In one of these abstract heaps, list and cur point to the same node $\overline{n}_1$, while in the other heap, they point to different nodes $\overline{n}_1$ and $\overline{n}_2$, respectively. The resulting abstract heap must represent both possibilities. Thus, in the resulting abstract heap, either list or cur must point to *two* non-null nodes.

When joining the precise abstract heaps, the heap analysis matches the nodes $\overline{n}_1$ and $\overline{n}_2$ such that in the resulting abstract heap, cur only points to one non-null node, while list points to two. This choice allows the heap analysis to match the most edges. This result is more useful for our analysis than the alternative. We want to avoid introducing disjunctive information for the cur variable which the loop uses to traverse the list.

Section 5.2 has shown that with the unmodified heap analysis, the result of the join is different, because the analysis did not remove edges. With the unmodified heap analysis, cur points to multiple non-null nodes in the resulting abstract heap.

6.6.6 Precise Abstract Heap at Fixed Point

Figure 22 depicts the abstract heap at the loop guard of `setListToZero` in the result of our analysis. This result contains sufficient information to extract a suitable loop invariant for the method.

Section 5.2 showed the result of the unmodified heap analysis for this method, and the problems that make the result unsuitable for specification inference. We will now explain how our analysis has addressed these problems.

Disjunctive Information The result of the unmodified heap analysis contained too much disjunctive information. The variable cur pointed to more than one non-null node. In the result of our analysis, cur only points to a definite node $\overline{n}_2$ or null. It is trivial to distinguish between these two cases in the specification. Our analysis has solved this problem by removing impossible edges, such that cur never points to more than one non-null node at the end of a loop iteration.

Cycles The abstract heap resulting from the unmodified analysis represented heaps with many possible cycles, which could not be specified with a recursive predicate. In the result of our analysis, the folded predicate instance at $\overline{n}_2$ guarantees that

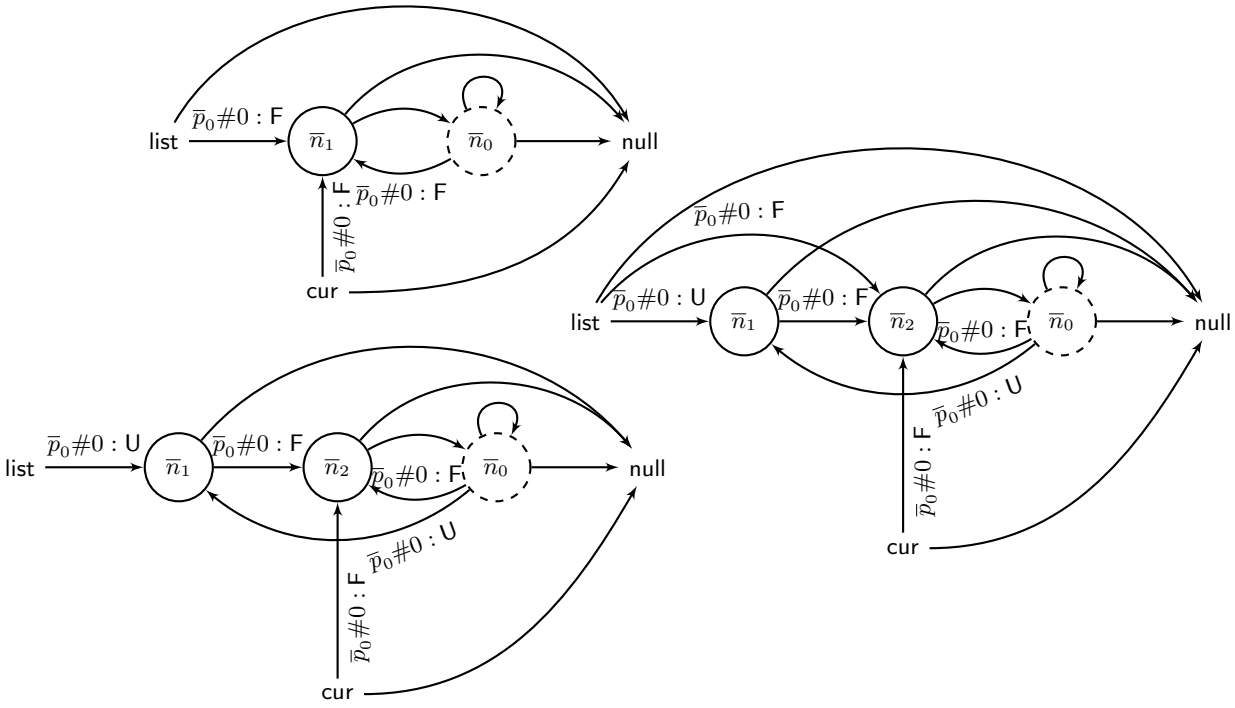


Figure 21: Abstract states before and after first loop iteration of `setListToZero` (left) and joined abstract heap (right). Each state contains $\overline{p}_0 \mapsto [\mathsf{val}, \mathsf{next} \to \{\overline{p}_0\}]$.

the data structure is acyclic. That is, starting from the object represented by $\overline{n}_2$, one must eventually reach null by recursing over next.

Unrelated Objects Finally, the summary nodes in the result of the unmodified heap analysis could also represent objects that do not belong to the list that the method recurses over. In the result of our analysis, the summary node $\overline{n}_0$ can still represent objects that are not part of the list. However, these objects are not a problem any more. When our analysis materializes a new definite node in the loop body for `cur.next`, then the analysis is certain that it belongs to the list. Thus, the analysis removes impossible edges to already unfolded nodes.

6.7 Object Allocation

In SIL, methods can allocate new objects by using a statement of the form `r := new()` for some local reference variable `r`. That is, the reference to the newly allocated object must always be directly assigned to local variable. The `new()` constructor does not take any arguments. Fields of the object need to be initialized separately.

In the heap analysis, allocating a new object and assigning it to a local variable or field are two separate operations. For simplicity, we will treat it as a single operation. In the abstract heap, a newly allocated object is represented by a fresh definite node $\overline{n}$. All reference fields of the newly allocated object are initialized to null. After the assignment

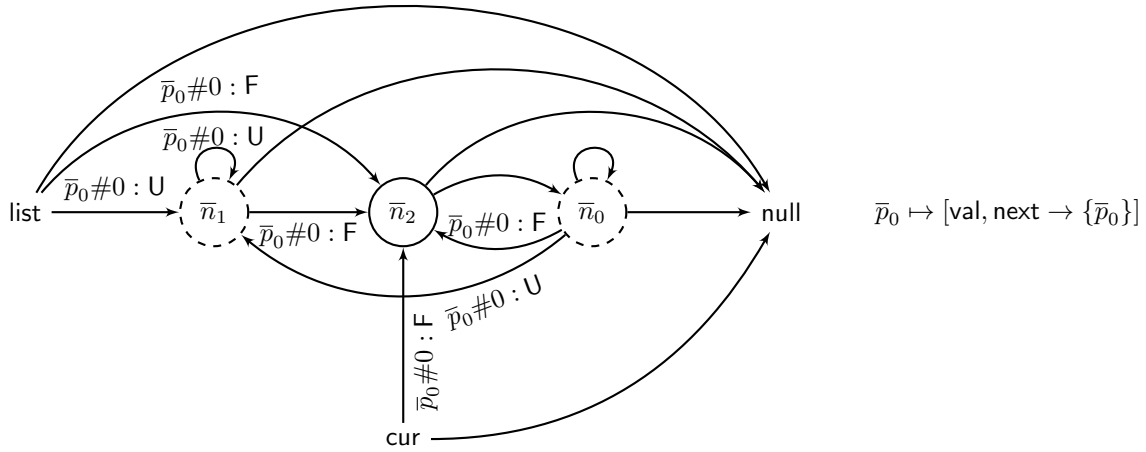


Figure 22: Abstract fixed point heap at the loop guard of `setListToZero`.

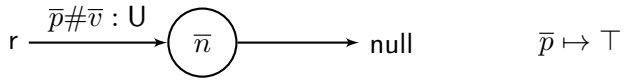


Figure 23: Part of abstract heap after object allocation statement `r := new()`.

to the local variable `r`, there will be a single edge $\overline{e}$ pointing to $\overline{n}$. For a formal description of the semantics of object allocation, refer to [5].

6.7.1 Introduction of Fresh Unfolded Predicate

In SIL, allocating an object gives the method permission to all fields of the object, independently of whether they are accessed or not. The object may be added to a data structure (see list generation example in Listing 5.2). Eventually, the object and the permission to its fields may be returned to the caller.

By default, the heap analysis would not add any Folded or Unfolded labels to $\overline{e}$. To the analysis, it would appear as if there were no permission to any field of $\overline{n}$. It would be cumbersome to remember which objects were allocated by the method itself and treat them in a special way.

It is possible to keep the treatment of objects uniform: the analysis creates a fresh predicate $\overline{p}$ and instance version number $\overline{v}$ and adds the label $\overline{p}\#\overline{v}$: Unfolded to the edge. The resulting abstract heap is shown in Figure 23. The Unfolded label is artificial in the sense that there was never an actual folded instance of $\overline{p}$ at the object.

The fresh predicate is $\top$ and subsequent accesses to fields of the object will populate $\overline{p}$ with permissions. This is a design choice to prevent fields from appearing in the predicate body that have nothing to do with the data structure the method is building. Our intuition is that SIL methods usually initialize all relevant fields of a newly allocated object.

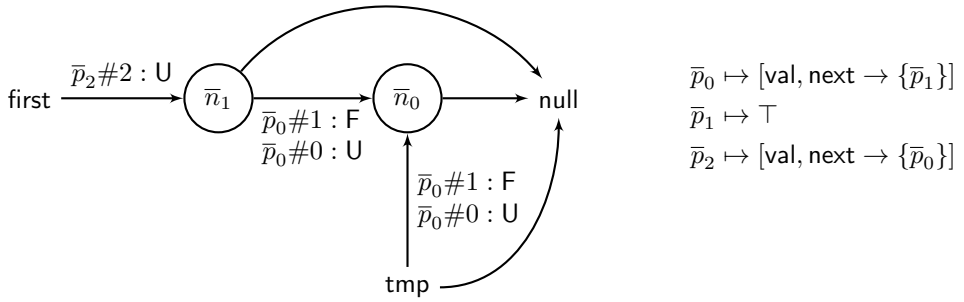


Figure 24: Abstract heap after the assignment `first.next := tmp` in the *second* analysis iteration of `firstNaturals`. The analysis adds $\mathsf{next} \to \{\overline{p}_0\}$ to the body of $\overline{p}_2$.

6.8 Field Assignments

Consider a SIL method that builds some data structure, such as `firstNaturals` in Listing 6. Such a method may assign an object that represents a part of the data structure to the field of an object that represents the whole data structure. The predicate that specifies the whole data structure often contains a nested predicate instance that specifies the assigned part of the data structure.

After a field assignment, the analysis adds such a nested predicate instance to the predicate that specifies the receiver object.

Example. In the loop body of `firstNaturals`, the field assignment `first.next := tmp` assigns the old head of the list (`tmp`) to the field `next` of the new head of the list `first`. Figure 24 depicts the abstract heap after this assignment in the *second* analysis iteration, representing the state in the first and second loop iteration.

Before that field assignment, the newly allocated $\overline{n}_1$ is described by a fresh predicate $\overline{p}_2$ with body [val] (see Section 6.7 on object allocation). At the same time tmp could either point to null (in the first loop iteration) or point to $\overline{n}_0$ (in the second loop iteration). $\overline{n}_1$ is specified with a folded instance of predicate $\overline{p}_0$ with permission to the fields val and next.

The field access `first.next` adds $\mathsf{next} \to \{\overline{p}_{fresh}\}$ to the body of $\overline{p}_2$ for a fresh predicate $\overline{p}_{fresh}$. Since the assigned node $\overline{n}_0$ is described by a folded instance of $\overline{p}_0$, the analysis merges $\overline{p}_{fresh}$ into $\overline{p}_0$. In the abstract heap, $\overline{p}_0$ is thus nested in $\overline{p}_2$. If $\overline{p}_{fresh}$ were not merged, the permissions to $\overline{n}_0$ would be lost after `tmp := null`.

6.9 Folding Predicate Instances

Suppose that the analysis joins two abstract states $\overline{\sigma}_1$ and $\overline{\sigma}_2$. One state may have a local variable with a folded predicate instance, while there is only an unfolded predicate instance for that variable in the other state.

To make the states consistent, the analysis eagerly folds predicate instances for local variables in both $\overline{\sigma}_1$ and $\overline{\sigma}_2$. These folds correspond to **`fold`** statements at the end of the basic blocks that the abstract states $\overline{\sigma}_1$ and $\overline{\sigma}_2$ originate from.

The following description only consider folding predicate instances whose argument is a local variable o. In future work, the approach could be generalized to allow the analysis to fold predicate instances with arguments such as $\mathsf{o}.f$.

Identifying Predicates for Folding. The analysis needs to decide whether to fold instances of any predicates or not. In order to fold an instance of a SIL predicate, all permissions in the body of the predicate need to be present. This includes permissions to fields, but also nested predicate instances.

For a local variable o, the analysis only considers a predicate $\overline{p}$ as a candidate for folding, if the following conditions are satisfied:

1. There are Unfolded labels for predicate $\overline{p}$ on the edges of o, but no Folded labels. In other words, the object that o points to was unfolded from an instance of $\overline{p}$ (and not folded again) or the object has been allocated by the method itself (see Section 6.7 on object allocation).

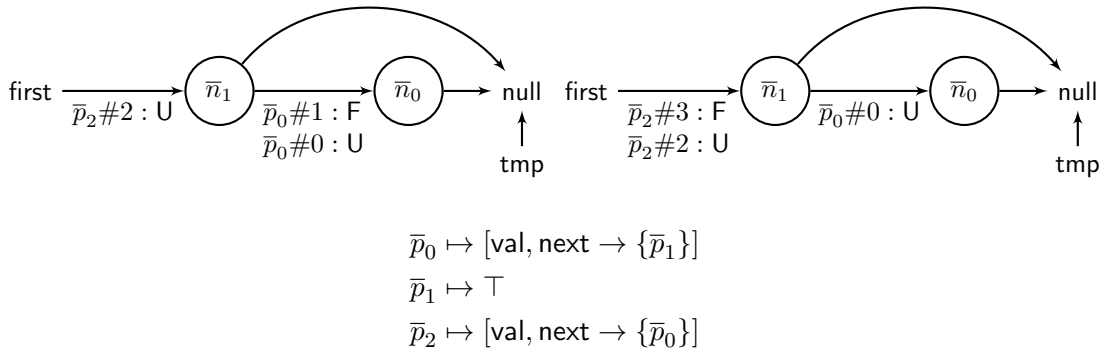
2. If the body of predicate $\overline{p}$ contains any nested predicate instances, then folded instances of these predicates need to be present in the abstract heap. Suppose that $\overline{p}$ contains a nested predicate $\overline{p}_{nested}$ for field f. Then, there must be a Folded label for $\overline{p}_{nested}$ on every edge that represents the field f or object o.

   The version numbers of these folded predicate instances do not matter. The analysis folds the instance of predicate $\overline{p}$ as long as there is any folded instances for each the nested predicates.

Adding New Folded Predicate Instance Labels. Let predicate $\overline{p}$ be a candidate for folding as described above. That is, there are labels $\overline{p}\#\overline{v}$: Unfolded on the edges of o. To express that the method now holds a folded instance of predicate $\overline{p}$ for o, a corresponding Folded label needs to be added to the edges of o.

The analysis uses a fresh version number $\overline{v}_{fresh}$ for the new $\overline{p}\#\overline{v}_{fresh}$: Folded predicate instance labels. The fresh version number allows the analysis to keep the existing $\overline{p}\#\overline{v}$: Unfolded label with the old version number. Keeping the Unfolded labels is important such that future unfoldings of predicate instances $\overline{p}\#\overline{v}$ can continue removing impossible edges to objects, including o, that were unfolded earlier.

Removing Folded Labels of Nested Predicate Instances. When folding $\overline{p}$, the analysis removes all Folded labels for nested predicate instances from the abstract heap. These predicate instances are now nested in the newly folded predicate instance. They are not available any more for other folding or unfolding operations.



$$\overline{p}_0 \mapsto [\mathsf{val}, \mathsf{next} \to \{\overline{p}_1\}]$$
$$\overline{p}_1 \mapsto \top$$
$$\overline{p}_2 \mapsto [\mathsf{val}, \mathsf{next} \to \{\overline{p}_0\}]$$

Figure 25: Abstract heap before (left) and after (right) folding the instance of predicate $\overline{p}_2$ for the local variable first at the end of the `firstNaturals` loop.

Example. Figure 25 illustrates the effect of folding an instance of predicate $\overline{p}_2$ for the local variable first at the end of the `firstNaturals` loop.

There is a label $\overline{p}_2\#2 : \mathsf{Unfolded}$ on the edge from first to $\overline{n}_1$. The body of $\overline{p}_2$ contains a nested instance of predicate $\overline{p}_0$, and there is a matching label $\overline{p}_0\#1 : \mathsf{Folded}$ on the edge from $\overline{n}_1$ to $\overline{n}_0$ for the field next. Thus, an instance of $\overline{p}_2$ can be folded for the variable first.

Folding adds the label $\overline{p}_2\#3 : \mathsf{Folded}$ to the first edge with the fresh predicate instance version number 3, but leaves the label $\overline{p}_2\#2 : \mathsf{Unfolded}$ label untouched. Additionally, the label $\overline{p}_0\#1 : \mathsf{Folded}$ is removed from the edge from $\overline{n}_1$ to $\overline{n}_2$.

Heuristics to Prevent Undesired Folding. Folding a candidate predicate $\overline{p}$ for some local variable o is not always desirable.

Consider the `setListToZero` example in Listing 7. The `list` variable always points to the first list element, while `cur` refers to the current list element. At the end of the first loop iteration, the analysis the local variable `list` a candidate for folding. However, folding the predicate instance for `list` would result in a loss of the folded predicate instance of `cur`, which already points to the second list element.

Our analysis uses two heuristics to prevent the folding of predicate instances that are likely to result in useless states. The analysis refrains from folding a candidate predicate instance for some local variable o,

1. if folding would cause a loss of any folded predicate instance for a different local variable, or
2. if o has more than one non-null edge.

Currently, the analysis folds predicate instances independently in two abstract states to be joined. A more holistic approach would be to take both abstract state into consideration, which could make the decision whether to fold or not more robust.

7 Extraction of SIL Specifications from Abstract States

The result of the analysis described in Section 6 provides an abstract state for each point in the analyzed method. The abstract states approximates the shape of the heap, values in the heap and with our extension, also predicates and predicate instances that describe data structures in the heap.

From these abstract heaps, we extract the following specifications and insert them into the original SIL program.

- We extract predicates from the predicate state at the end of the method.
- For each loop in the method, we extract a SIL assertion from the abstract state that represents the invariant of that loop. The extracted assertion contains predicate instances that are present in the abstract state, as well as assertions on variables and fields of objects.
- We extract the method's postcondition from the abstract state at the end of the method. The extraction is analogous to loop invariants.
- Extracted preconditions only contain instances of inferred predicates for reference parameters of the method. Since Sample currently does not support a backwards analysis, we cannot infer more preconditions, such as constraints on values.
- We add **`fold`** and **`unfold`** statements to the SIL program at the points in the program where a corresponding operation was performed by the analysis itself.

In the following, we will first describe how predicates are extracted (Section 7.1), and then the actual assertions (Section 7.2). Finally, we will describe how **`fold`** and **`unfold`** statements are extracted (Section 7.3).

7.1 Predicates

The representation of predicates in the analysis (see Section 6.2.1) allows for a simple, direct translation to SIL predicates. The identifier of the predicate in the analysis serves as the name of the resulting SIL predicate. We use `this` as the name of the predicate parameter. For each field with permission in the inferred predicate, a corresponding field access predicate can be inserted in the body body of the resulting predicate.

We extract predicates from the exit abstract state of the method. The analysis never removes permissions from predicates as a result of method operations or joins. Thus, the predicates in the method's exit state contain all the permissions that were added at various points of the method.

Example. Suppose that the predicate state contains the predicate $\overline{p}_0 \mapsto [\mathsf{val}, \mathsf{next} \to \{\overline{p}_0\}]$. Listing 9 shows the SIL predicate extracted from this representation.

```
predicate p0(this: Ref)
{
  acc(this.next) && acc(this.val) &&
  (this.next != null ==> acc(p0(this.next)))
}
```

Listing 9: Predicate extracted from representation $\overline{p}_0 \rightarrow [\mathsf{val}, \mathsf{next} \rightarrow \{\overline{p}_0\}]$.

7.1.1 Placeholder Predicates

The resulting predicate state may contain predicates that do not contain any permissions. For example, if the body of a method never accesses any field of a reference parameter, then the analysis never adds any permissions to the placeholder predicte associated with that parameter. Placeholder predicate are not useful for the specification, so they are ignored during extraction. It is sound to remove such placeholder predicates, because without any permissions, the body of an inferred predicate is `true`.

7.1.2 Predicate Aliases

The analysis may merge multiple predicates into a single predicate during join and widening (see Section 6.5). For example, the analysis may merge the predicates $\overline{p}_0$ and $\overline{p}_1$ into $\overline{p}_0$. In the state resulting from the join, the identifier $\overline{p}_1$ will not occur any more. Hence, no separate predicate is extracted for $\overline{p}_1$.

However, there may be abstract states at program points before the predicate $\overline{p}_1$ is merged into $\overline{p}_0$. When extracting assertions from such an intermediate abstract state, $\overline{p}_1$ should refer to the predicate eventually extracted for $\overline{p}_0$.

The analysis solves this problem with a global map that stores such predicate aliases. This is sound because predicate identifiers are globally unique and are never reused.

7.1.3 Inlining Shallow Predicates

The analysis may infer predicates that do not contain any nested predicates other than placeholders. An example of such a predicate is $\overline{p}_0 \mapsto [\mathsf{val}]$. We call such predicates *shallow*.

Instead of extracting actual instances of shallow predicates, we inline the bodies of such predicates. For example, consider that there is a folded instance of the above predicate $\overline{p}_0$ at the object referred to by `this`. Then, instead of the actual predicate instance `acc(p0(this))`, the resulting specification directly uses the field access predicate `acc(this.val)`.

The resulting specification is more concise as it contains fewer inferred predicates, and no `fold` and `unfold` statements are required for such predicates.

```
var val: Int
var next: Ref

method setListToZero(list: Ref)
{
    var cur: Ref
    cur := list
    while (cur != null)
    {
        cur.val := 0
        cur := cur.next
    }
}
```

Listing 10: SIL method, that sets all elements of a linked list to zero.

7.2 Assertions

In the following, we explain our approach to extracting a SIL assertion from a single abstract heap. The extraction involves several challenges because the representation of abstract heaps is not directly amenable for extracting SIL assertions.

We will illustrate the extraction of assertions for the `setListToZero` method shown in Listing 11. From the abstract states at the beginning of the method and at the loop guard, we extract a precondition and loop invariant, respectively.

7.2.1 Splitting Abstract Heaps to Handle Disjunctive Information

As explained in Section 3.2.5, a definite node $\overline{n}$ in an abstract heap may have multiple out-going edges labeled with the same field f. In any concrete heap, only one of the ambiguous edges labeled with field f of a source node $\overline{n}$ may exist. Ambiguous edges allow the analysis to track disjunctive information, i.e., to represent different possible heap shapes. An assertion extracted from an abstract heap should hold for all represented heaps, and the assertion may thus need to distinguish between them.

Consider the example of inferring the precondition for `setListToZero` in Listing 11. Figure 26 depicts the abstract state at the beginning of the method (on the left-hand side) that we want to extract the precondition from. The edges from list are ambiguous. There is only an instance of predicate $\overline{p}_0$ under the condition that list points to an object represented by $\overline{n}_0$. In the extracted assertion, we need to distinguish between the two cases.

Splitting Abstract Heaps. We make the problem of extracting an assertion from such an abstract heap more manageable by splitting the abstract heap into multiple abstract heaps that do not contain ambiguous edges any more. Then, we can extract an assertion from each of these abstract heaps and combine these assertion into a disjunction.

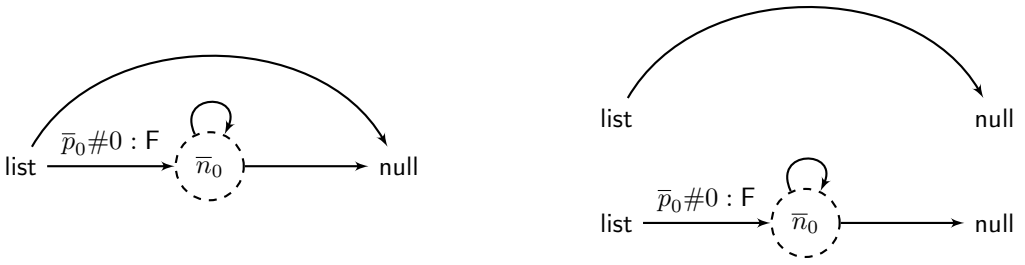


Figure 26: Splitting an abstract heap (left) into two abstract heaps (right). The states contain the predicate $\overline{p}_0 \mapsto [\mathsf{val}, \mathsf{next} \to \{\overline{p}_0\}]$.

Concretely, if there is a definite node $\overline{n}$ in the abstract heap with multiple out-going edges labeled with the same field f, then we create a separate abstract heap for each of these edges. In each of the resulting abstract heaps, there is only one edge labeled with f going out of $\overline{n}$. This splitting of abstract heaps happens recursively, because there may be multiple cases of ambiguous edges in the abstract heap.

The abstract state on the left-hand side of Figure 26 is split into two abstract heaps in which edges from list are not ambiguous any more. The upper abstract heap gives rise to the assertion `list == null`. In the lower abstract heap, there is certainly a predicate instance `acc(p0(list))`. These two assertion can be combined to the disjunction `(list == null) || acc(p0(list))`.

Avoiding Disjunctions in Assertions. Disjunctions are problematic in SIL because they must not contain access predicates such as `acc(p0(list))` (see Section 2.2). However, Access predicates may occur on the right-hand side of implications.

To avoid disjunctions, we try to extract an implication for each split abstract heap that contains the extracted assertion on the right-hand side. The left-hand side of the implication is a *sufficient condition* for that split abstract heap.

The sufficient conditions for the two abstract heaps in Figure 26 are simple, namely `this == null` and `this != null`, respectively. Combining the sufficient conditions with the extracted assertions, we get `(list == null) ==> (list == null)` and `(list != null) ==> acc(p0(list))`, respectively. The former impliciation is a tautology and thus does not need to be present in the extracted specification. That is, `(list != null) ==> acc(p0(list))` is the precondition extracted for `setListToZero`.

It is not always possible to find such sufficient conditions, especially if the abstract conditions of ambiguous edges intersect. For example, the variable list in the resulting abstract heap at the loop guard of `setListToZero` in Figure 22 has three edges. The extraction cannot distinguish between these three cases. Thus, the extraction does not split the heap for these edges and the resulting assertion does not contain list.

Currently, the extraction only supports sufficient conditions that test for nullness and non-nullness. Due to time constraints, we leave the support for a wider class of sufficient conditions as a future work.

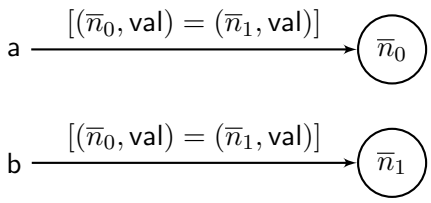


Figure 27: Abstract heap illustrating the extraction of value expressions.

7.2.2 Predicate Instances

A predicate instance state in abstract edge conditions is used to extract predicate instances for the resulting SIL assertion. For definite nodes, it is important not to extract multiple instances of the same predicate, even if multiple local variables point to that node.

The analysis eagerly folds predicate instances at the end of basic blocks (see Section 6.9). Thus, if there is an Unfolded label but no Folded label for a predicate instance at some object, it means that the analysis did not fold the predicate instance, for example because some nested predicate instances are missing. Thus, the extraction only considers predicate instances labeled as Folded.

7.2.3 Logical Assertions

It is possible to extract assertions on variables and fields from abstract heaps. For non-reference fields, the heap analysis uses identifiers such as (Trg, f) (see Section 3.2.1), that cannot be used in SIL assertions.

In a split abstract heap, the local variables do not have ambiguous edges any more. That is, in the abstract condition of an edge from a local variable o, we can soundly replace an identifier (Trg, f) with a field access expression $\mathsf{o}.f$.

Example. In the abstract heap in Figure 27, the variable a and b unambiguously point to $\overline{n}_0$ and $\overline{n}_1$, respectively. Thus, we can replace the identifiers $(\overline{n}_0, \mathsf{val})$ and $(\overline{n}_1, \mathsf{val})$ with the field access expressions `a.val` and `b.val`, respectively. This allows us to extract the assertion `a.val == b.val` from the abstract heap.

7.2.4 Postcondition

SIL postconditions may only refer to parameters and return values of the method, but not local variables. When extracting a postcondition from the abstract exit state of a method, all local variables therefore need to be removed from the state.

```
var val: Int
var next: Ref

method setListToZero(list: Ref)
    requires (list != null) ==> acc(p0(list))
{
    var cur: Ref
    cur := list
    while (cur != null)
        invariant (cur != null) ==> acc(p0(cur))
    {
        unfold acc(p0(cur))
        cur.val := 0
        cur := cur.next
    }
}

predicate p0(this: Ref) {
    acc(this.val, write) && acc(this.next, write) &&
    ((this.next != null) ==> acc(p0(this.next), write))
}
```

Listing 11: SIL method with inferred specifications, that sets all elements of a linked list to zero.

7.3 `fold` and `unfold` Statements

The analysis may unfold a predicate instance in the abstract state when a field access occurs (see Section 6.3). Similarly, the analysis eagerly folds predicate instances in abstract heaps that are joined (see Section 6.9). These operations performed by the analysis directly translate into corresponding **`fold`** and **`unfold`** in the SIL program with inferred specifications.

For the extraction, we check for each statement in the program if the analysis has unfolded any predicate instances while interpreting that statement. For any such predicate instance, a corresponding **`unfold`** statement is added before that statement.

Analogously, for every join of abstract exit states of two basic blocks, we detect what predicate instances where folded in each of these abstract states. For each such predicate instance, a **`fold`** statement is added to the end of the corresponding basic block.

Listing 11 shows the `setListToZero` method with all inferred specifications. The specification includes an **`unfold`** statement at the beginning of the loop, due to the subsequent field access `cur.val`.

Limitations.

- The extraction only considers folding and unfolding operations occuring at the

fixed point of the analysis. There is no guarantee that a exactly the same folding and unfolding operations have occurred in every iteration of the analysis.

As a result, there may be programs for which the analysis infers **`fold`** or **`unfold`** statements that will not verify. Future improvements to the analysis may enforce that the same folding and unfolding operations happen in every iteration.

- A CFG may contain basic blocks that do not allow the insertion of **`fold`** and **`unfold`** statements. For example, there is a separate basic block for the guard of a loop. If the analysis performs unfolds a predicate instance due to a field access in a loop guard, it is not possible to add a corresponding **`unfold`** statement to the SIL program.

8 Further Technical Work

8.1 Web Interface

Even for simple methods, the result of a heap analysis consists of sizable data structures that are difficult for humans to interpret. The result contains a complete abstract heap graph for every point in the program. Furthermore, an abstact heap consists of nodes, edges between them and abstract value, predicate and predicate instance states on *every* edge.

Implementing and debugging an analysis that operates on such deep data structures is a daunting task and requires optimal debugging facilities. It is cumbersome to explore an abstract heap with the debugging facilities of a modern IDE, because they do not account for the graphical nature of heaps.

Sample contains a Java-based GUI that can visualize the control flow graph (CFG) of a method and the individual abstract heaps in the analysis result. The GUI is based on Java's GUI toolkit Swing[2] and the graph drawing library JGraphX[3]. The existing GUI has several drawbacks:

- The GUI does not make it possible to easily navigate between states at different points in the program. For instance, it would be desirable to see what effect a statement has on the abstract state by navigating to that abstract state. Furthermore, the GUI always opens a new window to display a selected aspect of the result, which soon leads to a cluttered workspace.
- It only displays the abstract state in the last iteration of the analysis, i.e., the resulting fixpoint. However, it is often necessary to learn how the analysis arrives at that fixpoint, especially if the fixpoint is unexpected.

To address these issues, a new web interface has been developed as a part of this project.

8.1.1 Functionality

The new web interface allows users to analyze test files and inspect the results, all in a single web browser window. Similar to the existing GUI, the web interface initially visualizes the CFG of the analyzed method (see Figure 28).

Test File Selection. The web interface automatically discovers test programs in provided folders and offers them for analysis. The files are even detected if they are inside of a JAR file. This capability should pave the way for distributing Sample, its web interface and test programs as a single JAR to interested parties.

[2] Java Swing: http://www.oracle.com/technetwork/java/architecture-142923.html
[3] JGraphX: https://github.com/jgraph/jgraphx

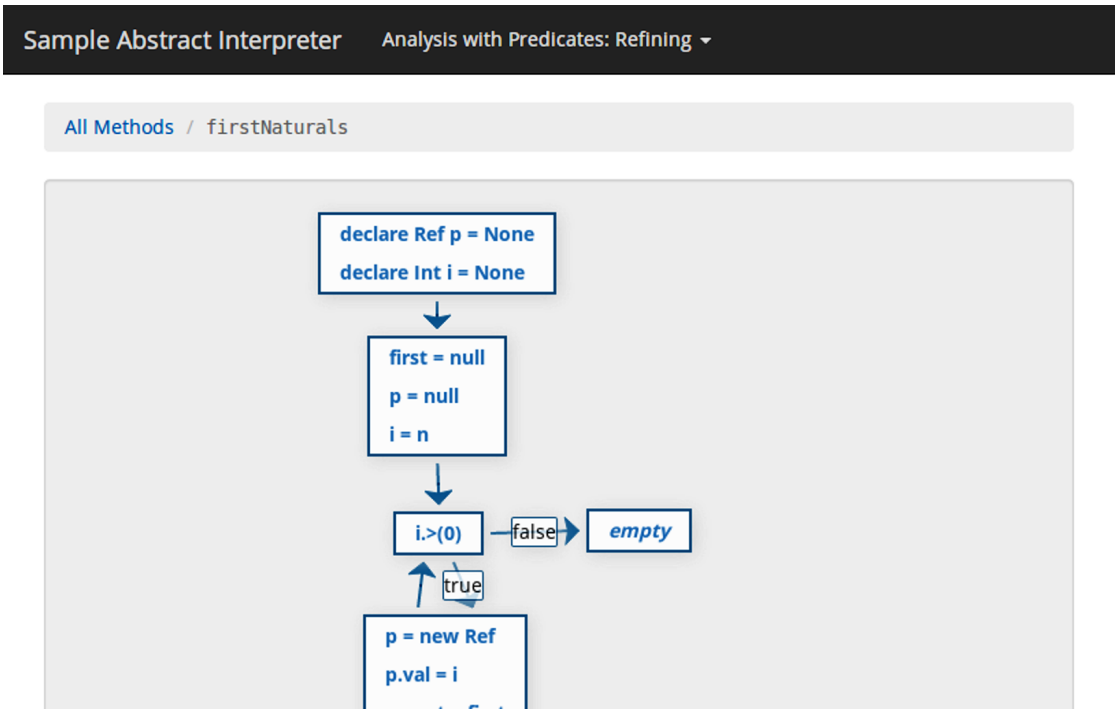


Figure 28: Web interface displaying the CFG of `firstNaturals` with clickable blocks.

Analysis Selection. The drop-down menu at the top of the web interface allows user to choose between different analyses or analysis configurations. In Figure 28, the program was analyzed with our predicate-aware heap analysis. An interested user could conveniently analyze the program with the unmodified heap analysis as well.

Navigation. Figure 3 shows the web interface when inspecting a single abstract heap. The breadcrumbs at the top inform the user about his or her current location. The buttons on the right-hand side allow the user to navigate to between states at consecutive points in the program.

Abstract State in Any Analysis Iteration. One of the most important additions is the ability to inspect abstract states of the program in earlier iterations of the analysis before a fixpoint was reached. In Figure 3, the buttons labeled 0 to 7 allow the user to navigate to the abstract state of any of the eight fixed point iterations at this program point.

In the abstract heap graph, the nodes can be dragged and dropped.

Abstract Edge States. Hovering the mouse pointer over an edge of the heap graph causes the abstract state on that edge to be displayed on the right-hand side. The abstract edge state may differ depending on the type of analysis. For our analysis, the panel displays both the abstract value state, as well as the predicate and predicate instance state.

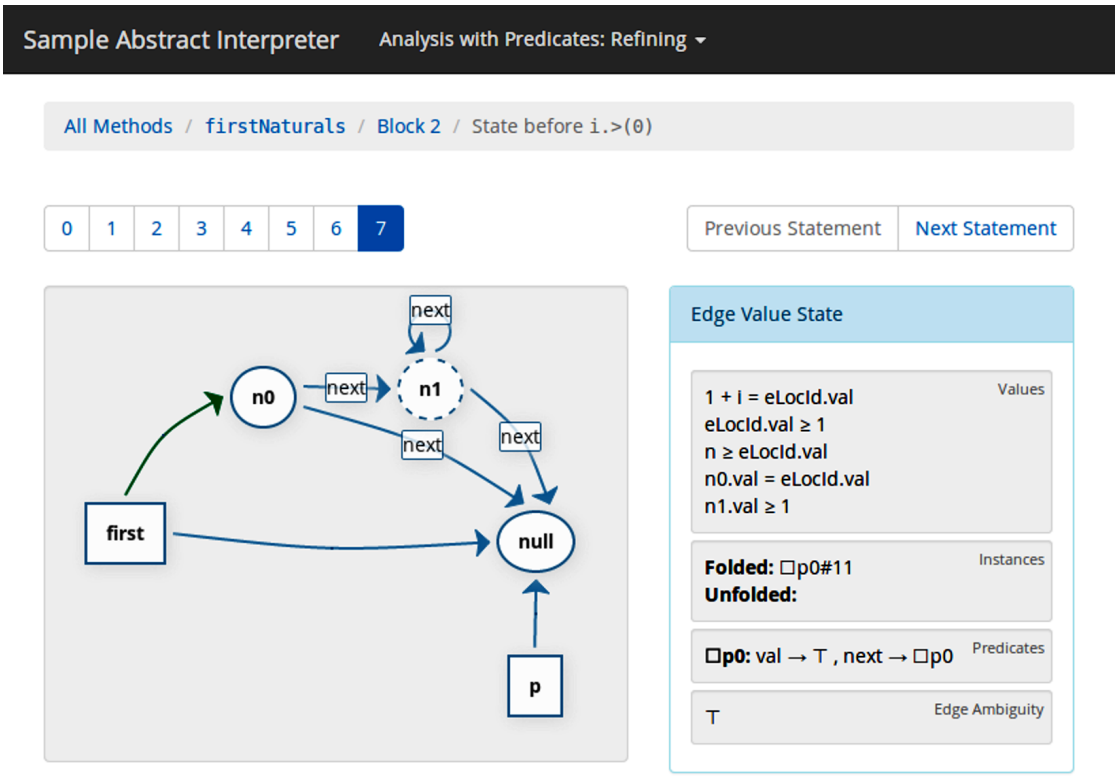


Figure 29: Web interface displaying the fixpoint, loop invariant abstract heap of `firstNaturals` and the state on the edge from first to $\overline{n}_0$.

8.1.2 Technical Basis

Sample's new web interface is written in Scala, JavaScript, HTML and CSS. The goal of the implementation was to keep the amount of custom server-side and client-side code low by reusing existing libraries wherever possible.

The web interface can be launched by running the main method of a class `App`. After launching it, the web interface is accessible at `localhost:8080` or another port number of choice.

Server-Side. Scalatra[4] serves as a light-weight webserver and web micro-framework.

To render pages of the web interface, we use Twirl[5], the templating engine used by the Play Framework[6]. Twirl seamlessly integrates with Scala and allows type-safe templating. Twirl templates contain a mix of HTML and Scala and are compiled into Scala source files by SBT and then treated like other Scala source files.

Listing 12 shows an example of such a template file that renders a HTML list of all nodes in a given heap graph. Assuming the template file is named `HeapGraphView.scala.html`, it is then possible to generate the HTML list from a heap graph by simply instantiating `html.HeapGraphView(heapGraph)`.

```
@(heapGraph: HeapGraph[_])
<ul>
@heapGraph.nodes.map { node =>
   <li>@node.name</li>
}
</ul>
```

Listing 12: Example Twirl template file `HeapGraphView.scala.html`.

Client-Side. To render interactive CFGs and heap graphs, the web interface uses the JavaScript library jsPlumb[7] based on jQuery[8] and jQuery UI[9].

jsPlumb is not capable of automatically finding a good layout for graphs. Therefore, the web interface further relies on the JavaScript library dagre[10] to compute a reasonable node layout. The library dagre is built for laying out DAGs, which neither applies to CFGs nor heap graphs in general. Hence, the layout is often inferior to layouts computed by sophisticated graph visualization software such as Graphviz[11].

[4] Scalatra: http://www.scalatra.org/
[5] Twirl: https://github.com/spray/twirl
[6] Play Framework: http://www.playframework.com/documentation/2.0/ScalaTemplates
[7] jsPlumb: http://jsplumbtoolkit.com/demo/home/jquery.html
[8] jQuery: http://jquery.com/
[9] jQuery UI: https://jqueryui.com/
[10] dagre: https://github.com/cpettitt/dagre
[11] Graphviz: http://www.graphviz.org/

For components such as panels and navigation elements, the web interface is based on Bootstrap[12]. Bootstrap is a popular front-end web framework for quickly building robust and feature-rich web applications.

8.2 Testing Infrastructure

From the beginning of the project, the goal was to automate the testing of the implementation as much as possible. Doing so manually would be very time-consuming.

8.2.1 Generalization of SIL's Testing Infrastructure

At the beginning of the project, Sample did not offer an automated end-to-end testing infrastructure to build upon. Instead of writing one from scratch, it was more reasonable to adapt the sophisticated testing infrastructure built for SIL verifiers.

SIL's end-to-end testing infrastructure locates all or specific SIL program files in designated resources directories and verifies them with one or more SIL verifiers. These SIL input programs can contain annotations that describe the expected output of the verifier. The testing infrastructure uses ScalaTest[13], which provides a convenient integration with IDEs such as IntelliJ IDEA.

As part of the project, we heavily generalized SIL's existing testing infrastructure [7]. Developers can now use it to test any system that processes source code files and produces outputs for specific positions in that source code. Annotations in the source code files describe the expected output.

The new infrastructure is agnostic with respect to the program language being used and to the type of output being produced. In fact, it is already being used for automated end-to-end testing of a backwards analysis [6] of TouchDevelop[14] programs with Sample.

8.2.2 Test Annotations

Without any annotations in the input program, the system under test is expected not to produce any output for that program. Any output will result in a test failure.

Annotations that specify the expected output have the form comments with a special prefix. The most important types of annotations are the following:

`//:: ExpectedOutput(key:value)` Such an annotation indicates that the system under test must produce the output `key:value` at the first non-annotation line following the annotation. The `key` designates the expected type of output. The `value` part is optional and can include further detail about the expected output. Examples include `not.wellformed:receiver.null` and `sample.assert.failed`.

[12] Bootstrap: http://getbootstrap.com/
[13] ScalaTest: http://www.scalatest.org/
[14] TouchDevelop: https://www.touchdevelop.com/

//:: UnexpectedOutput(key:value, /<project>/issue/<n>/) The system under test may have a known bug or limitation that results in unexpected output. To prevent the test suite from failing perpetually because of this known and reported issue, the `UnexpectedOutput` annotation causes the output to be ignored. Once the corresponding issue has been resolved, the annotation can be removed.

//:: MissingOutput(key:value, /<project>/issue/<n>/) Analogously, a known problem may prevent a certain output from being produced. The `MissingOutput` annotation prevents the test case from failing until it is resolved.

8.2.3 Translation and Heap Analysis End-to-End Testing

We test the correctness of the translation of SIL CFGs to Sample CFGs indirectly by analyzing the translated CFG with Sample, without trying to infer specifications. The SIL test programs contain assertions. For each assertion, we output an error if the assertion may not always be satisfied according to the overapproximating abstract state of the program at that point. In other words, we use Sample as a SIL "verifier" that only supports logical assertions.

For every SIL language feature that we support in the translation, there is a SIL program such as the following:

```
var r: Ref
r := new()
assert(r != null)

//:: ExpectedOutput(sample.assert.failed)
assert(2 < 0)
```

In addition to the translation test cases, there are also 26 regression test cases for issues that have been discovered and resolved in Sample's heap analysis.

8.2.4 Specification Inference End-to-End Testing

It is more challenging to test that our analysis infers the desired specification for a given SIL method. Test annotations as described in Section 8.2.2 are not suited for this task because they check outputs purely based on strings.

Instead, we rely on the SIL verifier Silicon. Only if the SIL method extended with inferred specifications verifies, is the test considered successful.

Calling Methods. The success criterion that the method verifies is not very strong. First, the inferred precondition may be too strong (e.g., `false`). In such a case, the method would always verify, but could not be called at all. Analogously, the inferred postcondition may be too weak. For example, the inferred postcondition may not return certain permissions to the caller.

To strengthen the test success criterion, we add a method `test` to every SIL test program that calls the method which we want to infer specifications for. The `test` method already has specifications and the test infrastructure does not apply specification inference to it.

Support for Predicates. Suppose that, for a SIL method, a recursive predicate should be inferred and the inferred precondition should contain a corresponding instance. The calling method could not satisfy that inferred precondition, because the inferred predicate does not exist yet in the original SIL program.

To solve this problem, we add a predicate to the SIL test program that matches exactly the predicate that should be inferred and use it in the calling method. The analysis itself is unaware of this predicate. However, the extraction logic will detect that a predicate in the original program matches the inferred predicate. That is, the inferred predicate contains exactly the same permissions and nested predicate instances. If this is the case, the existing predicate will be reused in the extended program, which will then verify.

Listing 31 shows an example of a SIL test program for a list traversal method.

```
var val: Int
var next: Ref

/** Method to infer specifications for. */
method traverse(list: Ref)
  requires (list != null) ==> acc(valid(list), write) // Inferred
{
  var cur: Ref
  cur := list
  while (cur != null)
   invariant (cur != null) ==> acc(valid(cur), write) // Inferred
  {
    unfold acc(valid(cur), write) // Inferred
    cur := cur.next
  }
}

/** Predicate that should be detected and reused when building
  * the extended program. */
predicate valid(list: Ref) { // Existing (but inferred and reused)
  acc(list.next, write) &&
  ((list.next != null) ==> acc(valid(list.next), write))
}

/** Tests that the precondition of traverse is not too strong.
  * Ignored by specification inference. */
method test(list: Ref)
  requires acc(valid(list), write) // Existing
{
  traverse(list)
}
```

Listing 13: SIL test program that contains both existing and inferred specifications.

9 Experimental Results

We implemented our approach to inferring SIL specifications in the static analyzer Sample and used the implementation to evaluate our approach. The implementation includes the translation of SIL programs (Section 4), our extension to the combined heap and value analysis (Section 6), and the extraction of specifications from the results of that analysis (Section 7).

For the evaluation, we use a set of SIL programs that operate on linked lists and binary trees, but also non-recursive data structures. We performed the experiments on a machine with an Intel Core i7-3667U CPU (2.0GHz) and 8 GB of RAM, running the Ubuntu 13.10 (64-bit). For the heap analysis, we use the Polyhedra [2] value domain implemented in Apron [8]. For all experiments, materialization (see Section 3.2.4) is enabled in the heap analysis and widening (see Section 3.2.3) is set to occur after three iterations of the analysis.

Table 1 lists the methods that we used for the evaluation, along with the amount of time required to infer specifications for them. The only method with pre-existing specifications is `firstNaturals`, whose precondition is `n > 0`. In the following, we will discuss the inferred specifications of the methods `copyContainer`, `reverseList` and `traverseTree`. The inferred specifications of our running examples `setListToZero` and `firstNaturals` are shown in Listing 7 and Listing 6, respectively.

Copying a Container. The method `copyContainer` in Figure 14 copies a container (an object with a `val` field) provided as a parameter, unless the parameter is **`null`**. Our analysis infers suitable pre- and postconditions that allow the method to verify.

This example highlights two features of the specification extraction:

- Since a container can be specified with a shallow predicate that only contains a field permission and no nested predicate instances, the extraction inlines the definition of the inferred predicate (see Section 7.1.3).

  In our implementation, the inlining of shallow predicates can be enabled or disabled. If disabled, the inferred precondition and postcondition of the `copy` method would contain predicate instances. In addition, the method body would contain one **`unfold`** and two **`fold`** statements.

- In addition to assertions involving permissions, the inferred postcondition also contains logical assertions such as `this.val == other.val`. These assertions strengthen the postcondition and thus make the method more useful to callers of the method. Our implementation currently only extracts logical assertions on fields of objects if they are framed by field access predicates.

Reversing a Linked List. The method `reverseList` in Listing 15 takes a linked list as argument, reverses it and returns the resulting list. The example shows that our analysis can infer specifications for a method whose state may contain more than one

Data Structure	Method	Inferred Specifications	Time
Container	`makeContainer`	Postcondition with permission to created container.	0.055
Container	`copyContainer`	Precondition with permission to original container, postcondition with permission to original and copied container.	0.161
Linked List	`setListToZero`	List predicate with instance in precondition and loop invariant. **`unfold`** statement.	0.856
Linked List	`firstNaturals`	List predicate with instance in loop invariant and postcondition. **`unfold`** statement.	0.607
Linked List	`reverseList`	List predicate with instances in precondition, loop invariant and postcondition. **`unfold`** and **`unfold`** statement.	1.319
Binary Tree	`traverseTree`	Binary tree predicate with instances in precondition and loop invariant. **`unfold`** statement.	4.554

Table 1: Inferred specifications and analysis time (in seconds) for individual methods.

```
var val: Int

method copyContainer(this: Ref)
    returns (other: Ref)
    requires (this != null) ==> acc(this.val)
    ensures (this == null) ==> (this == other && other == null)
    ensures (this != null) ==> (
      acc(this.val) && acc(other.val) &&
      (this.val == other.val) && (other != null))
{
    if (this == null) {
        other := null
    } else {
        other := new()
        other.val := this.val
    }
}
```

Listing 14: SIL method with inferred specifications, that copies a given container, unless it is **`null`**.

data structure at once, each of which needs to be described with a recursive predicate instance. The analysis also correctly infers both an **unfold** and **fold** statement.

The example also illustrates that abstract heaps with ambiguous edges are split recursively to extract assertions (see Section 7.2.1). In the abstract state that we extract the loop invariant from, both `oldList` and `newList` could either be **null** or refer to a list element. The extraction splits this abstract heap into four abstract heaps in which both `oldList` and `newList` are unambiguous, and extracts an assertion from each of these heaps. The sufficient condition of one of these heaps is for example `newList != null && oldList == null`. Our implementation simplifies the resulting assertion such that for example, `tmp == null` becomes unconditional, since it holds in all split abstract heaps. However, the implementation did not detect that `(oldList != null) ==> acc(p0(oldList))` holds independently of `newList`. This is only a concern for human interpretability of the specification, but not the verifier.

Note that the inferred predicate `p0` does not contain the permission for the field `val`. Since the method never accesses the field `val` of any element in the list, a precondition that only supplies permission to the `next` field of each list element is strong enough.

Traversing a Binary Tree. The method `traverseTree` in Listing 16 shows that our analysis also generalizes to recursive data structures other than linked lists. The method traverses a binary tree and requires a predicate that recurses over two fields `left` and `right`. The method verifies with the inferred specification, consisting of a precondition, loop invariant and **unfold** statement in the loop body.

```
var next: Ref
var val: Ref

method reverseList(list: Ref) returns (newList: Ref)
  requires (list != null) ==> acc(p0(list))
  ensures (newList != null) ==> acc(p0(newList))
{
  var oldList: Ref
  var tmp: Ref
  oldList := list
  newList := null
  tmp := null
  while (oldList != null)
    invariant tmp == null
    invariant (newList != null) ==>
      acc(p0(newList)) &&
      ((oldList != null) ==> acc(p0(oldList)))
    invariant (newList == null) ==>
      (oldList != null) ==> acc(p0(oldList))
  {
    tmp := oldList
    unfold acc(p0(oldList))
    oldList := oldList.next
    tmp.next := newList
    newList := tmp
    tmp := null
    fold acc(p0(newList))
  }
}

predicate p0(this: Ref) {
  acc(this.next) &&
  ((this.next != null) ==> acc(p0(this.next)))
}
```

Listing 15: SIL method with inferred specifications, that reverses a linked list.

```
var left: Ref
var right: Ref
var val: Int

method traverseTree(tree: Ref, key: Int)
  requires (tree != null) ==> acc(p0(tree))
{
  var node: Ref
  node := tree
  while (node != null)
    invariant (node != null) ==> acc(p0(node))
  {
    unfold acc(p0(node))
    if (key < node.val) {
      node := node.left
    } else {
      node := node.right
    }
  }
}

predicate p0(tree: Ref) {
  acc(tree.val) &&
  acc(tree.left) &&
  acc(tree.right) &&
  (tree.left != null) ==> acc(p0(tree.left)) &&
  (tree.right != null) ==> acc(p0(tree.right))
}
```

Listing 16: SIL method with inferred specifications, that traverses a binary tree.

10 Conclusion

We have presented a novel approach to inferring specifications for SIL programs via extending a recent combined heap and value analysis [5].

We have shown that the results of the heap analysis do not provide sufficient information to extract specifications for programs that operate on recursive data structures. For such a program, it is necessary to identify suitable predicates that specify the data structures. To overcome this challenge, we have refined the heap analysis such that it incrementally infers information on predicates and predicate instances, based on how the program operates on the heap.

We have implemented our approach in the static analyzer Sample. The evaluation has shown that our approach can infer specifications for programs that operate on linked lists and trees, without requiring any user-provided annotations.

Specification inference has the potential to reduce the effort required for verification and could thus lead to a wider adoption of verification in the industry. We hope that our humble contribution inspires further research in the area of specification inference.

10.1 Future Work

There are several directions for further work.

Extracting More Complex Logical Assertions. Our extraction of assertions from abstract heaps focuses on predicate instances that allow the method to verify. However, the abstract heaps may also include relationships among values on the heap.

The current implementation only extracts simple logical assertions for the values on the heap. More complex assertions, possibly involving recursive abstraction functions, could make inferred specifications more useful. The extraction of such assertions would be technically challenging, but could exploit the inferred information on predicates.

Lifting Restrictions on Predicates. We have imposed restrictions on predicates that allows for a simple representation of predicates in the analysis. We believe that these restrictions can be weakened with more engineering effort, thus enabling the inference of more expressive predicates.

For example, our mechanisms for making the heap analysis more precise in the presence of recursive predicate instances is expected to work for predicates with more than one parameter, as long as the recursion only occurs over the fields of one parameter.

Adding Support for Magic Wand Operator. The inferred specification for the method `setListToZero` does not return the permissions for the list to the caller of the method. Preserving the permissions with the features of SIL described in Section 2 is involved. The recently added support for the magic wand operator $-\!\!*$ to the SIL verifier Silicon [12]

promises less overhead for the specifications of loops. In order to extract specifications that use the magic wand operator, it may be necessary to track additional information in the analysis such as the variable that points to the whole list. However, Unfolded labels may already serve as valuable information.

10.2 Acknowledgements

Special thanks go to my advisors Dr. Alex J. Summers and Milos Novacek for many long and fruitful discussions, providing new perspectives on the problems at hand and valuable feedback on the written report. Furthermore, I would like to thank Prof. Dr. Peter Müller for giving me the opportunity to work on this thesis at the Chair of Programming Methodology. Last but not least, thanks go to my family for their encouragement.

References

[1] Bernhard Brodowsky. Translating Scala to SIL. Master's thesis, ETH Zurich, 2013.

[2] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[3] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 269–282, New York, NY, USA, 1979. ACM.

[4] Pietro Ferrara. Static type analysis of pattern matching by abstract interpretation. In *Proceedings of the 12th IFIP WG 6.1 international conference and 30th IFIP WG 6.1 international conference on Formal Techniques for Distributed Systems*, FMOODS'10/FORTE'10, pages 186–200, Berlin, Heidelberg, 2010. Springer.

[5] Pietro Ferrara, Milos Novacek, and Peter Müller. Automatic inference of heap properties exploiting value domains. 2013.

[6] Raphael Fuchs. Inferring counterexamples from abstract error states. Master's thesis, ETH Zurich, 2014.

[7] Stefan Heule, Ioannis T. Kassios, Peter Müller, and Alexander J. Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. In *Proceedings of the 27th European conference on Object-Oriented Programming*, ECOOP'13, pages 451–476, Berlin, Heidelberg, 2013. Springer.

[8] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 661–667, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] Christian Klauser. Translating Chalice into SIL. Bachelor's thesis, ETH Zurich, 2012.

[10] K. Rustan Leino and Peter Müller. A basis for verifying multi-threaded programs. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 378–393, Berlin, Heidelberg, 2009. Springer-Verlag.

[11] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 247–258, New York, NY, USA, 2005. ACM.

[12] M. Schwerhoff and A. J. Summers. Lightweight support for magic wands in an automatic verifier. Technical Report 0, ETH Zurich, 2014.

[13] Malte Schwerhoff. Symbolic execution for Chalice. Master's thesis, ETH Zurich, 2010.

[14] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 148–172, Berlin, Heidelberg, 2009. Springer.

[15] Alexander J. Summers and Sophia Drossopoulou. A formal semantics for isorecursive and equirecursive state abstractions. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 129–153, Berlin, Heidelberg, 2013. Springer-Verlag.

A SIL Language Definition

The following syntax of SIL in BNF has been typeset by Stefan Heule [7] and Bernhard Brodowsky [1].

$,^*$ is a special repetition operator that denotes a comma-separated list.

```
      sil-program ::= ( domain | field | function | predicate | method )*

           domain ::= "domain" domain-name "{"
                          domain-function*
                          axiom*
                      "}"

      domain-name ::= ident | ident "[" ident ,* "]"

  domain-function ::= "unique"? "function" ident "(" formal-arg ,* ")" ":"
    type

       formal-arg ::= ident ":" type

            axiom ::= "axiom" ident "{"
                          exp ";"?
                      "}"

            field ::= "var" ident ":" type

         function ::= "function" ident "(" formal-arg ,* ")" ":" type
                          precondition*
                          postcondition*
                      "{" exp "}"

     precondition ::= "requires" exp ";"?
    postcondition ::= "ensures" exp ";"?
        invariant ::= "invariant" exp ";"?

        predicate ::= "predicate" ident "(" formal-arg ,* ")" "{" exp "}"

           method ::= "method" ident "(" formal-arg ,* ")" ("returns" "("
             formal-arg ,* ")")?
                          precondition*
                          postcondition*
                      "{" local-decl* stmt "}"

       local-decl ::= "var" ident ":" type

             stmt ::= (stmt ";"?)*
                    | "assert" exp
                    | "inhale" exp
                    | "exhale" exp
                    | "fold" "acc" "(" loc-access "," exp ")"
                    | "unfold" "acc" "(" loc-access "," exp ")"
                    | loc-access ":=" exp          // field assignment
                    | ident ":=" exp               // local variable
                      assignment
                    | "if" "(" exp ")" "{"
```

```
                stmt
            "}"
            ("elsif" "(" exp ")" "{"
                stmt
            "}")*                       // (any number of
              elseif branches)
            ("else" "{" stmt "}")?      // (the else branch is
              optional)
          | "while" "(" exp ")"
                invariant*
            "{" stmt "}"
          | ident ":=" "new()"          // object creation
          | ident "(" exp,* ")"         // method call
          | ident,* := ident "(" exp,* ")"  // method call with
            return values
          | "goto" ident                // goto statement
          | ident ":"                   // a label (can be used
            as target for goto)
          | "fresh" (ident*) "{"        // fresh abstract read
            permission block
                stmt
            "}"

exp ::= exp "?" exp ":" exp             // conditional expression
      | exp "==>" exp                   // implication
      | exp ("||" | "&&") exp           // disjunction and
        conjunction
      | "!" exp                         // boolean negation
      | exp ("==" | "!=") exp           // equality comparison
      | exp ("<" | "<=" | ">" | ">=") exp // ordering (both
        numerical/permission)
      | exp ("+" | "-") exp             // math operators (both
        numerical/permission)
      | exp "*" exp                     // int/int, perm/perm
        and int/perm multiplic.
      | exp ("\ " | "%") exp            // integer division and
        modulo
      | ("+" | "-") exp                 // math operators (both
        numerical/permission)
      | ident "(" exp,* ")"             // function application
      | loc-access                      // field read or
        predicate
      | integer                         // integer literal
      | "null"                          // null literal
      | "true" | "false"                // boolean literal
      | ident                           // local variable read
      | "result"                        // result literal in
        function postconditions
      | "acc" "(" loc-access "," exp ")"   // accessibility
        predicate
      | "forall" formal-arg,* "::" trigger,* exp   //
        universal quantification
      | "exists" formal-arg,* "::" exp   // existential
        quantification
      | "(" exp ")"
      | "[" exp "," exp "]"             // inhale exhale
        expression
```

```
                | "perm" "(" loc-access ")"   // current permission
                  of given location
                | "write"                     // full permission
                  literal
                | "none"                      // no permission literal
                | "epsilon"                   // epsilon permission
                  literal
                | "wildcard"                  // wildcard permission
                | exp "/" exp                 // concrete fractional
                  permission

                | "Seq" "[" type "]" "()"     // the empty sequence
                | "Seq" "(" exp,* ")"         // explicit sequence
                | "[" exp ".." exp ")"        // half-open range of
                  numbers
                | exp "++" exp                // sequence append
                | "|" exp "|"                 // length of a sequence
                | exp "[" exp "]"             // sequence element for
                  given index
                | exp "[" ".." exp "]"        // take the some of the
                  first elements
                | exp "[" exp ".." "]"        // drop some elements
                  at the end
                | exp "[" exp ".." exp "]"    // take and drop at the
                  same time
                | exp "in" exp                // element containment
                  test
                | exp "[" exp ":=" exp "]"    // sequence with one
                  element updated

                | "Set" "[" type "]" "()"     // the empty set
                | "Set" "(" exp,* ")"         // explicit set
                | "Multiset" "[" type "]" "()"// the empty multiset
                | "Multiset" "(" exp,* ")"    // explicit multiset
                | "|" exp "|"                 // set/multiset
                  cardinality
                | exp "union" exp             // set/multiset union
                | exp "intersection" exp      // set/multiset
                  intersection
                | exp "setminus" exp          // set/multiset
                  subtraction
                | exp "subset" exp            // set/multiset subset
                  test

   trigger ::= "{" exp,* "}"                  // a trigger for a
     quantification

loc-access ::= exp "." ident                  // field access
             | ident "(" exp,* ")"            // predicate

      type ::= "Int" | "Bool" | "Perm" | "Ref"  // primitive types
             | "Seq" "[" type "]"             // sequence type
             | ident                          // type variable or
               non-generic domain type
             | ident "[" type,* "]"           // generic domain type

     ident ::= "[a-zA-Z$_][a-zA-Z0-9$_']*" // an identifier
```

(specified as regular exp)