A Strict-Size Logic for Featherweight Java Extended with Update

Rody Kersten

Master’s Thesis Computer Science
Radboud University Nijmegen
Thesis Nr 597, December 2009

Supervisors
Prof. dr. M.C.J.D. van Eekelen
Dr. O. Shkaravska
Drs. A. Tamalet
Dr. ir. E. Poll (reader)
Abstract

Bounding heap space usage can be of critical importance in embedded applications and will become more relevant to regular computer users because of the advent of cloud computing, where users have to pay for the server resources they use. It can also play an important role in security analysis, mainly for protection against denial-of-service attacks. The use of heap space depends on the sizes of the stored datastructures. Size analysis of datastructures is therefore a prerequisite to heap space analysis.

In the AHA project polynomial size analysis has been developed for a functional language. Annotations that specify the output-on-input size relations of functions can be verified. These annotations can either be given by the programmer or automatically inferred.

This work partially transfers the results of the AHA project to an imperative object-oriented language. The chosen language is Featherweight Java Extended with Update (FJEU), which is an extended subset of Java. Annotations in JML-style are introduced with which the sizes of parameters post method execution can be specified as polynomials over the sizes pre execution. A program logic is defined which can be applied to verify these annotations and a proof of its soundness is sketched. This strict-size logic successfully deals with side-effects and aliasing, issues that are specific to imperative languages. A crucial part of the strict-size logic is the frame rule, which is adapted from separation logic. This creates the possibility to reason about a specific part of the heap without paying attention to the rest. Its soundness in combination with the logical rule for field update is proved. Furthermore, the use of the strict-size logic is clarified by a series of examples and a proof of its soundness with respect to the operational semantics of FJEU is sketched.
Contents

1 Introduction 9

1.1 Static Analysis ............................................. 9
1.2 Resource Analysis ......................................... 9
1.3 Heap Space Analysis ..................................... 10
1.4 Size Analysis ............................................... 10
1.5 Polynomial Size Analysis for an Imperative Language ... 11
1.6 Related Work ............................................... 11
  1.6.1 Sized Types and Memory Regions ...................... 11
  1.6.2 Amortized Heap Space Usage Analysis (AHA) .......... 12
  1.6.3 MRG, MOBIUS and Embounded ......................... 13
  1.6.4 Polynomial Quasi-Interpretations ...................... 14
  1.6.5 Other Work .............................................. 15
1.7 Paper Outline ............................................. 15

2 Choice of Language 17

2.1 Featherweight Java ......................................... 17
2.2 Featherweight Java Extended with Update .................. 19
2.3 Resource Aware Java (RAJA) ............................... 21
2.4 Size Aware Java (SAJA) .................................... 21
2.4.1 JML ............................................ 21
2.4.2 Size Annotations .............................. 23

3 Program Logic for Strict-Size Analysis .............................. 25
3.1 Formal Concepts ........................................ 26
3.1.1 Class Ontology .................................... 26
3.1.2 Fields ........................................... 26
3.1.3 Aliasing and Reachability ......................... 27
3.1.4 Sized Types ....................................... 29
3.1.5 Conditions ......................................... 32
3.1.6 Methods .......................................... 33
3.1.7 The Ω System of Equations ...................... 34
3.1.8 Fetching Field Sizes ............................ 35
3.2 Axiomatic Semantics .................................... 35
3.2.1 Basic Expressions .................................. 35
3.2.2 Let-Binding ....................................... 36
3.2.3 Sequencing ........................................ 37
3.2.4 Method Invocation ............................... 37
3.2.5 Field Access ...................................... 38
3.2.6 Field Update ....................................... 38
3.2.7 Casting ........................................... 39
3.2.8 Conditionals ...................................... 40
3.2.9 Return Statements ............................... 40
3.2.10 Frame Rule ....................................... 41
3.3 Alias Control .......................................... 44
3.3.1 Harmless Conditional Aliasing .................... 45
3.3.2 Verifying That All Aliasing is Harmless ........... 47
CONTENTS

4 Examples of Annotation Verification Using the Strict-Size Logic 49

4.1 List Concatenation ........................................... 49

4.2 Tree to List Transformation ................................. 56

4.2.1 Annotation Correctness for the leafToList() Method ... 56

4.2.2 Annotation Correctness for the nodeToList() Method ... 59

4.2.3 Application to the Main Method ......................... 68

4.3 List Reversal .................................................... 70

4.4 Polynomial Example .......................................... 76

5 Soundness Proof Sketch 83

5.1 Relating the Strict-Size Logic to the Operational Semantics ... 83

5.2 Soundness of the Update/Frame Rule ....................... 87

5.3 Sketch of the Complete Proof ................................. 90

6 Evaluation and Future Work 91

6.1 Limitations of the Program Logic ............................ 91

6.1.1 Conditional Sizes ........................................ 91

6.1.2 Better Treatment of Aliasing .............................. 92

6.1.3 Size is a Single Number .................................. 93

6.1.4 Other Restrictions ........................................ 93

6.2 Other Extensions .............................................. 93

6.2.1 Full Soundness Proof ...................................... 93

6.2.2 Implementation ............................................ 94

6.2.3 Size Inference ............................................. 94

6.2.4 Expansion to Java .......................................... 94

7 Conclusions 95
Chapter 1

Introduction

Qualitative analysis of software is an increasingly popular subject among computer scientists. Because more and more software is used ubiquitously, sometimes in possibly life-threatening situations, validation of its correctness has gained interest among researchers and popularity among its producers. A single fault, shortage in memory or too slow performance can cost lives in certain applications. Therefore methods must be derived to verify that software will perform as expected in any situation.

1.1 Static Analysis

One form of judging the quality of software is by using static analysis. This encompasses analyzing a program without executing it (as opposed to dynamic analysis). A lot of interesting properties of a program can be derived statically, although completely proving correctness of a program is undecidable, due to the halting problem. A well known static analyzer is the compiler. In addition to verifying that the program matches the language syntax and semantics, and type-checking, most current compilers also apply, for instance, data-flow analysis to eliminate dead code. Other properties of software that might be derived with static analysis include liveness, deadlock, soundness and completeness properties (with model-checking for example), computational complexity and partial correctness.

1.2 Resource Analysis

Computer systems often have limited resources available, especially when embedded systems or systems with an exceptionally intensive task are considered. To assure quality and reliability of these systems, different forms of resource analysis
have been invented to make estimates of (upper bounds to) the usage of a re-
source by a program. An important aspect of the reliability of programs is their
vulnerability to (denial-of-service) attacks, for which it is important to know where
resources are consumed.

Another development which strengthens the need for accurate resource analysis is
the advent of cloud computing, in which a certain amount of computer resources
is rented by a company or an end-user.

Resources under consideration include time, energy and memory. Ideally, one
system would be applicable to various types of resources. However, since the field
is still relatively unexplored, most analyses apply to one specific resource only.

1.3 Heap Space Analysis

The resource in focus for this work is heap space. Together with an analysis for
stack space (which is more straight-forward in general) this would provide an upper
bound on the memory usage of programs. This can be useful in situations where
limited memory is available or where the memory that is made available depends
on the application.

Heap space analysis is the focus of the Amortized Heap space Analysis (AHA)
project [47], on which the research presented herein will build.

Good estimations of heap space usage can be of particular importance in embedded
situations or for cloud computing (because in both cases limited resources are
available) and in security analysis, because there it is important to know where
memory leaks may occur.

1.4 Size Analysis

The use of heap space is determined by the size of the data stored in the heap,
which in turn depends on the structure and size of the data structures used. It
was therefore found that an analysis of the impact of functions (methods) on the
sizes of these data structures is prerequisite to heap space analysis.

This type of analysis was deemed size analysis and focuses on the relation between
the sizes of the input and the sizes of the output of functions. It was thus far
developed to handle polynomial output-on-input relations for a simple functional
language.
1.5 Polynomial Size Analysis for an Imperative Language

In this work, the transfer to a basic imperative, object oriented language is made. This yields several new challenges, mainly side effects and aliasing. To stay close to another line of research on heap space analysis [24, 23, 31], the language chosen is Featherweight Java Extended with Update (FJEU), as defined in [24]. This language is, as the name implies, an extension of Featherweight Java [28]. FJEU will be extended with JML-like annotations to create a Size Aware Java.

To handle side effects a Hoare-style program logic is used instead of a type-system, which was used in previous research on size analysis. Because this program logic implements a polynomial analysis of strict sizes we refer to it as the strict-size logic.

1.6 Related Work

Several lines of research exist that consider heap-space consumption estimation. However, most research has been focused on establishing monotonic bounds, for example in the various articles based on sized types and memory regions described in Section 1.6.1. In the AHA project [47], described in detail in Section 1.6.2, polynomial size analysis has been developed, but for a functional language. The research described herein focuses on transferring polynomial size analysis to an imperative language. Other relevant research has been done under the flag of the MRG project [41] and its successors, which is described in Section 1.6.3 and on polynomial quasi-interpretations, which are discussed in 1.6.4. Related work that does not fall into any of these categories is discussed in Section 1.6.5.

1.6.1 Sized Types and Memory Regions

An approach that is often taken in the field of memory usage estimation is the use of sized types, as initially proposed by Hughes and Pareto [27], who add size annotations to the type system. Apart from execution in constant space, their programming language Embedded ML also guarantees program termination. Termination analysis is also the focus of [11] and [2], for which size expressions based on ordinal variables, ordinal successor and an ordinal limit are used. These simple size expressions are sufficient for termination analysis, but for heap space analysis more complex expressions are needed.

Portillo et al. use a sized type and effect system in [39] and also develop a size inference algorithm, which makes their approach useful for the analysis of programs that are not annotated with sizes. Sized types are used by Grobauer and Bernd to derive upper bounds on execution times of programs in terms of the size of their
input in [20]. Vasconcelos and Hammond develop an inference system for cost equations in [49]. However, only monotonic bounds are studied and recurrence relations are left unsolved.

A type system for memory usage verification of object-oriented programs is presented by Chin et. al. in [14], which may be used to calculate conservative worst-case bounds.

1.6.2 Amortized Heap Space Usage Analysis (AHA)

The Amortized Heap Space Usage Analysis (AHA) project [47] is aimed at developing a methodology for determining bounds on heap space consumption. Like in most of the aforementioned research, sized types are used to express relations between input and output sizes.

Amortization is a term which has its origins in the financial world, where it is used to express that multiple smaller payments account for a certain larger amount. The term was applied to a brand of complexity analysis in [46] and to heap space analysis in [24]. Here it means that certain operations that increase the potential of a data structure may compensate for other operations that decrease its potential, resulting in a simpler estimation of the change in potential of the total.

In [44, 43], a technique was developed for checking the correctness of user-specified output-on-input size relations on first-order functions. This technique improved on existing methods, because it was able to determine polynomial bounds on sizes rather than monotonic bounds. A user of this system can annotate the types of a function with the sizes of the input arguments and the return type. An example of this is a function that flattens a list of lists. In the functional language used, lists are defined (implicitly) as:

\[
\text{List}(\alpha) = \text{Nil} \mid \text{Elem}(\alpha, \text{List}(\alpha))
\]  

(1.1)

So a list is a number of Elem constructors, each containing an element of type \( \alpha \), with a Nil at the end. The size of a list may simply be expressed by its length, which is equal to the number of Elem constructors. The flatten function takes a single argument: a list of length \( n \), containing \( n \) lists of size \( m \). It outputs a single list containing all the elements, which thus has a size of \( n \times m \). Such bounds may be expressed by polynomials, which allows for more complicated functions to be considered than functions with linear size dependencies.

An algorithm for automatically inferring these size relations was introduced in [44] and described in detail in [43]. Size inference amounts to guessing the polynomials that describe the relation between input and output sizes. The algorithm works by generating hypotheses for polynomials of an increasing degree. It is proved in [43] that the generated hypotheses are always correct if the size polynomial is indeed
of the considered degree. The algorithm terminates if and only if the size relation can actually be expressed by a polynomial and the type can be checked (which may not always be the case due to type-checker incompleteness). In practice, an upper bound can be given on the maximal degree of polynomials to consider. The algorithm depends on the execution of (parts of) the object program for the generation of hypothetical polynomials. This may be another source of non-termination.

Since up to this point a very restricted functional language with only integer and list data types was used, support for algebraic data types was added, proving the systems ability to handle more complex programs \cite{45}. Whereas the size of a list can be described by a single integer representing its length, it is harder to do this for more complicated data types, such as trees:

\[
\text{Tree}(\alpha) = \text{Leaf} | \text{Node}(\alpha, \text{Tree}(\alpha), \text{Tree}(\alpha))
\] (1.2)

It could be decided to count just leafs, or just nodes, but in either case information would be lost. Also, in some cases the size of the output can depend on just the number of leafs, or just the number of nodes, or both (think of a function that replaces each leaf in a tree with a node with an element and two leafs, this will double the number of leafs and increase the number of nodes with the number of leafs on input). Therefore it was decided to store a size tuple instead of a single integer. A user may however specify a special size function for an algebraic data type, which leaves the possibility to, for example, express the size of a list by just its length.

Finally, the approach was adapted to work with both upper and lower bounds on the output-on-input size dependencies \cite{42}. Some functions yield an output of a variable size, dependent on their input. An example of this is a function that deletes the first occurrence of an element in a list, if it exists. If the length of the input list is \( n \), this yields a list of size \( n - 1 \) if the element occurs in the list, or \( n \) if it does not. This further increased the domain of functions that can be analyzed following this approach.

A demonstration program implementing the methodology is available at [www.aha.cs.ru.nl](http://www.aha.cs.ru.nl). Amortization techniques have not been developed for this approach yet, since a complete polynomial size analysis system is prerequisite to that. The research proposed in this document aims at transferring the results of the AHA project to an object-oriented imperative language.

1.6.3 MRG, MOBIUS and Embounded

In the Mobile Resource Guarantees (MRG) project \cite{41} and its follow-ups MOBIUS\textsuperscript{1} and Embounded \cite{21}, several different approaches to heap-space consump-
CHAPTER 1. INTRODUCTION

Static analysis have been developed. These projects are mainly focused on proof carrying mobile code.

In [23], Hofmann and Jost describe a technique to statically predict the heap space usage of first order functional programs. In their approach, every variable has an initial potential and any usage of heap space must be paid for from this potential, implying that the potential of the initial configuration establishes an upper bound on the needed heap space. Their approach differs from that taken in the AHA project in its dependence on linear size relations, whereas we allow polynomial relations. However, amortization has not been developed for our approach yet.

In [24], Hofmann and Jost extend their approach to a simple object oriented imperative language. They describe the FJEU language (based on Featherweight Java, which is a functional subset of Java) and the resource aware RAJA language, based on FJEU. Their RAJA heap consumption model is based on a freelist (of free “memory units”) and amortization is used to determine an upper bound on the required size of this list. This imperative variant of their technique is further developed in the to-be-published [26].

The functional branch of their research is further extended in [31]. It now utilizes the Schopenhauer functional language and can be used to estimate the usage of various resources, including heap space, stack space and time complexity.

The Hofmann-Jost analysis is adapted by Campbell to infer upper bounds on stack space in his PhD thesis [13].

The approach presented by Albert et al in [3] and [4] is based on two-phase cost analysis, the first phase being the construction of recurrence relations, capturing the costs of a program in terms of the size of its input data, while the second phase is the conversion of these relations into closed form. In [4] this method is presented for Java byte code, while [3] is more focused on the second phase, since a framework for establishing upper bounds of the recurrence relations is presented.

A bit different direction is taken in [7] and [8], where a resource aware program logic for Grail is discussed. Grail [9] is an intermediate language in the compilation of Camelot [35] and O’Camelot [50] to Java byte code. It can be seen as a functional subset of the Java Virtual Machine Language. The same amortization technique is used as in [24]. This creates the possibility to write resource aware programs for any device capable of executing Java byte code, such as mobile telephones and smartcards.

1.6.4 Polynomial Quasi-Interpretations

Another related line of research is that on polynomial quasi-interpretations [10, 6, 5, 19]. Here static analysis is done by assigning a numerical function, compatible with the computational semantics, to each program symbol. This is usable for multiple resources like time and space. For size analysis with polynomial quasi-
interpretations, a data structure is seen as a monotonic polynomial extended with
the max operation. For instance, a list can be seen as $T + 1$, where $T$ is a numerical
variable abstracting the length of the tail. Monotonic polynomial upper-bounds
can be obtained using such interpretations.

The main differences with the work described herein are that polynomial quasi-
interpretations have only been developed for functional languages and that only
monotonic bounds can be derived. For non-monotonic bounds polynomial quasi-
interpretations may lead to significant over-estimations.

1.6.5 Other Work

In [29], Jay and Cockett present shapely types, which separate data from structure,
and introduce shape polymorphism. Jay and Sekanina study shape checking of
array programs in [30]. In this work, the (exact) size relations between input
and output variables are given as a program expression over sizes. However, the
derivation of an arithmetic function from this program expression is beyond the
scope of their work.

Braberman, Garbervetsky and Yovine present a memory consumption analysis for
an imperative object-oriented language in [12]. Their analysis is based on the
number of object allocations, as opposed to the sizes of data structures, i.e. they
study the number of constructor calls as a (polynomial) function of (integer)
inputs. When the number of constructor calls depends on the size of an input
data structure, their approach relies on loop bounds analysis. Also, recursion can
not be handled using their method.

1.7 Paper Outline

The imperative language of choice is described in Chapter 2, including a description
of the JML-type annotations. The strict-size logic that can be used to verify these
annotations is presented in Chapter 3, which is the main contribution of this paper.
Some examples of annotation verification using the strict-size logic are given in
Chapter 4 and in Chapter 5 a soundness proof is sketched. The approach is
evaluated and suggestions for future work are described in Chapter 6. Chapter 7
concludes the work.
Chapter 2

Choice of Language

The language under consideration for this analysis is Featherweight Java Extended with Update (FJEU). This language was described by Hofmann and Jost in [24] and is indeed an extension of the Featherweight Java language [28]. The language is extended to Resource Aware Java (RAJA) by adding heap consumption annotations, also in [24]. In this thesis, the FJEU language will be extended to Size Aware Java (SAJA) by adding size annotations. This relation is depicted in Figure 2.1.

![Language ontology](image)

Figure 2.1: Language ontology

Featherweight Java, on which FJEU is based, will be discussed in Section 2.1. The same semantics are shared by FJEU, RAJA and SAJA, since all the annotations are removed from the code before interpretation. This semantics and the various extensions made by FJEU will be discussed in Section 2.2. The RAJA language will be briefly discussed in Section 2.3 and the SAJA language (i.e., the size annotations used in this paper) will be introduced in Section 2.4.

2.1 Featherweight Java

Featherweight Java was described in [28] by Igarashi, Pierce and Wadler. It is a thoroughly simplified core calculus of the Java language, which omits all but the
mere basics. According to the authors, Featherweight Java relates to Java in the
same way the lambda-calculus relates to functional languages like ML or Haskell.
It is meant to supply a means for easily proving properties of the Java language
without the overhead of more complex structures.

Just five forms of expressions exist in Featherweight Java: object creation, method
invocation, field access, casting and variables. The concept of assignment has been
omitted entirely, thereby preventing methods from having side effects. Feather-
weight Java is thus a functional core of Java. The omission of an assignment
operation also ensures the absence of aliasing. Other features of Java that have
been omitted from Featherweight Java include (but are not limited to) basic types
(int, bool, etc.), null pointers, concurrency, overloading and abstract method decla-
rations.

Some example Featherweight Java code implementing a linked list is shown in
Listing 2.1. A list is either an \texttt{Elem} with a link to the next element of the list, or it is a \texttt{Nil}, the end of the list. An \texttt{append} method is defined, which concatenates two lists by replacing the \texttt{Nil} at the end of the list with the start of \texttt{l}.

In this example, object creation occurs at line 27 (the \texttt{new} statement). On the same line, the \texttt{this.next.append(l)} expression is a method invocation. In the \texttt{append} function for \texttt{Elem}, \texttt{this.a} is a field access and \texttt{l} is a variable. The newly constructed \texttt{Elem} on line 27 is implicitly cast to a \texttt{List} when the function returns. Objects can also be explicitly cast as follows: \texttt{(List) new Nil()}.

Other remarkable things to notice are the fact that a supertype is always given, even if it is \texttt{Object}, the constructors are always written out, even though they have a standard form (calling \texttt{super} first, then storing all the fields), and this is always explicitly stated.

There are no abstractly declared methods in Featherweight Java, which means that for the function \texttt{append} to exist for every \texttt{List}, it needs to be declared in the \texttt{List} class as a dummy version. The idea behind this is that in reality it will only be called on objects of either the \texttt{Nil} or the \texttt{Elem} type. There is however no way to express this in the language.

For a full syntax and semantics the reader is referred to [28], where the language is also shown to be effective for its purpose by extending it to a version including generics. Several properties of this generic version of Featherweight Java are proved, demonstrating that such proofs can be done fairly easy.

### 2.2 Featherweight Java Extended with Update

Featherweight Java was extended to FJEU in [24]. FJEU stands for Featherweight Java Extended with Update. However, this name can be a little misleading. Even though field update is the most important extension, other features were also added or modified. FJEU also shows some similarity to the Classic Java language by Flatt et al. [18]. The complete syntax and semantics of FJEU are given in the unpublished manuscript [25], along with a soundness proof of the RAJA type system.

With the addition of field update the purely functional nature of Featherweight Java is lost in FJEU, because in FJEU methods can have side effects on their parameters and the fields of the class that they belong to. It also introduces aliasing, because now references to existing objects, instead of only newly constructed ones, can be assigned to fields.

In Featherweight Java the default way in which objects are constructed is to pass all the fields as arguments to the constructor method and store them. Since objects and their fields are immutable because of the absence of an field update operator, this is the only possible way. In FJEU, the only permitted constructor is
the default, empty constructor. All the fields can then be set using assignments.

FJEU permits only upcasts, which means that if we define the class ontology relation $D <: C$ as $D$ being a subclass of $C$, then a cast $(C)D$ is only valid if $D <: C$. Null pointers are permitted and may be cast to any class. Explicit extension of a superclass as in Featherweight Java is no longer obligatory. The basic data type int and the notion of void as a return type were added. An if-then-else construct was added, but in a pattern-matching-like form, since there are no booleans in the language.

```java
class List {
    Object elem;
    List next;

    void append(List l) {
        let List n = this.next in
        if n instanceof List then
            let List _ = n.append(l) in
            return
        else
            let List _ = this.next <- l in
            return;
    }
}
```

Listing 2.2: Example FJEU code

The structure of an if-then-else construct is shown in Listing 2.2 on line 7. The if-statement always checks if an object is an instance of a certain class. This is implicit in the semantics, because boolean type expressions do not exist.

The example implements a linked list in FJEU. Here, a null pointer is used to mark the end of the list, instead of a Nil object, allowing the code to be compressed to a single class. Pattern-matching is used to distinguish between the cases where this.next is null and where it is a pointer to the next element of the list. The lists are appended by replacing the null pointer which marks the end of the list by a reference to $l$.

Assignment takes the form $C.a \leftarrow b$, as on line 11 of Listing 2.2. FJEU assignment differs from its implementation in full Java, because the aforementioned expression evaluates to the updated version of $C$, not to $b$ as in Java. ML-like let-binding is also added to the language, as shown on lines 6, 8 and 11, which is another source of aliasing (apart from field update).
2.3 Resource Aware Java (RAJA)

FJEU is further extended in [24] to RAJA by adding heap consumption annotations. These annotations are based around the idea that a type has a certain potential, from which uses of heap space have to be accounted for. The total of this potential then forms an upper limit to the needed heap space.

Each class is annotated with at least one view. The combination of a class with a view is called a RAJA type or a refined type. Views are used to be able to associate a single class with multiple potentials. Each refined type is then annotated with its potential (an integer). Each attribute of a RAJA class has two views: the get view and the set view, of which the set view is intended to be stronger. Methods are annotated with the views of their parameters and return type, and two integers, \( p \) and \( q \). The potential of the parameters will be consumed, plus an additional potential of \( p \). After the method is executed a potential of \( q \) is gained (methods may use heap space, then free it up again). Also, the return type may carry an additional potential.

The main difference with the approach presented herein is that RAJA may be used to determine a linear upper bound on heap space, whereas polynomial bounds can be handled by the size logic presented herein.

2.4 Size Aware Java (SAJA)

In this paper, instead of the RAJA heap annotations, size annotations are added to FJEU, creating Size Aware Java (SAJA). Because a Java-like language is considered, JML-like annotations are chosen. JML is a widely adopted standard for annotations in Java programs and various tools exist to statically check these annotations (for example ESC/Java2).

2.4.1 JML

The acronym JML stands for Java Modeling Language [34, 33]. It is a specification language, made for design-by-contract purposes. Programs are annotated with Hoare logic style pre and postconditions and invariants. The annotations are enclosed between the special symbols */@ and @*/ on a single line after the // symbol. Therefore they are interpreted as comments by Java compilers, retaining the ability to use any desired compiler.

\[ \text{http://secure.ucd.ie/products/opensource/ESCJava2/} \]
An example of JML annotated code is shown in Listing 2.3. The annotations in this example are used to ensure that the content of the glass stays between zero (empty) and MAX_CONTENT (full), and also to check that both methods are correctly used. Preconditions are used on lines 15 and 21, postconditions on lines 9, 16 and 22 and an invariant is specified on line 6. The field content needs to be explicitly declared assignable for each method that needs to assign a new value to it. Various tools exist that can be used to check that these conditions hold, of which ESC/Java2 is the most widely known. Because JML is a widely adopted standard in the Java community and it is based on Hoare logic (which we intend to use in our strict-size logic) it is an ideal fit for our annotations.

Heap Space Usage Constraints in JML There is some support in JML for annotations concerning upper bounds on the used heap space through the space and workingspace expressions, which can be used, respectively, to retrieve the amount of heap space (in bytes) used by an object, and to specify the maximum amount of heap space that will be used by a method call or explicit constructor invocation. It is obvious that this is too limited for our approach, which is focused
on size analysis rather than heap space constraints. So, a JML-like syntax of our own invention will be used here.

### 2.4.2 Size Annotations

In this section the size annotations are described by an example and a formal definition is given.

```java
class List {
    Object elem;
    /*@ count @*/ List next;

    /*@
    @ requires (\sizedtype l; List{pl})
    @ && (\sizedtype \this; List{pt})
    @
    @ ensures (\sizedtype l; List{pl})
    @ && (\sizedtype \this; List{pl+pt})
    @
    @ mayaffectaliasing \this; l
    @*/
    void append(List l) {
        let List n = this.next in
        if n instanceof List then
            let List _ = n.append(l) in
            return
        else
            let List _ = this.next <- l in
            return;
    }
}
```

Listing 2.4: Example SAJA code

As can be seen in Listing 2.4, there are annotations for fields and for methods. Both types of annotation are in JML style.

Because we only want to calculate the size of the list structure itself, the field next is annotated to be counted. The field elem is implicitly not counted. A more thorough (formal) definition of size is given in Section 3.1.4. Here it suffices to understand that the size of a list is given by the size of next plus one, i.e. the length of the list.

The syntax of this annotation is simple: if a field is counted into the size of an object, the annotation /*@count@*/ is added right before the definition of the
field. If this annotation is not present the field is not counted.

The second type of annotation specifies the pre and postconditions of methods, regarding sized types. The pre and postconditions are given in the JML requires and ensures clauses, respectively. The precondition in the example (Listing 2.4, lines 6 and 7) specifies that the size of the parameter \( l \) before execution of the method is \( pl \). The size of \( this \) before execution of the method is \( pt \). The postcondition (lines 9 and 10) specifies that after appending list \( l \), the size of \( l \) is unchanged and the size of \( this \) is equal to \( pl + pt \).

The \texttt{sizedtype} predicate is defined as:

\[
sizedtype := (\texttt{sizedtype object; \, classname}\{p\})
\]

\[
object := \texttt{\result} \mid \texttt{this} \mid P
\]

\[
classname := \texttt{String}
\]

\[
p := \texttt{int} \mid \texttt{SizeVar} \mid p + p \mid p - p \mid p \ast p
\]

where \( P \) is a parameter name and \texttt{SizeVar} is a size variable.

The annotation \texttt{may affect aliasing \this; l} indicates the possibility that this function changes the aliasing relation between \texttt{this} and \( l \). What this means exactly will be made clear in Section 3.3.

The syntax of the \texttt{may affect aliasing} annotation is defined as:

\[
\texttt{may affect aliasing} := \texttt{may affect aliasing\ object; \ object}
\]
Chapter 3

Program Logic for Strict-Size Analysis

In this chapter a program logic for sized types in FJEU is proposed, the strict-size logic. Where in the previous work on size analysis [43, 48, 45, 42] a type system was used, the choice of a form of Hoare logic [22] was more appropriate here. The reason for this is that while in a purely functional language there are only expressions that evaluate to a certain value or expression, which is always the same, in an imperative language expressions or statements may have side effects. An imperative language is therefore more focused on the notion of state. Because Hoare logic is based on the state before and after a statement or expression, this can handle side effects and is a better fit for imperative languages.

Two data structures will be used in this chapter to illustrate the given concepts and definitions. The first is the implementation of a list, as given in Listing 2.4 in the previous chapter. The second example is an implementation of a tree structure, which is given in Listing 3.1.

```java
1 class Tree {
2     Object elem;
3 }

4 class Node extends Tree {
5     /*@ count @*/ Tree left;
6     /*@ count @*/ Tree right;
7 }
```

Listing 3.1: Tree structure in FJEU

In the program logic specified in this chapter we make use of a series of formal concepts, which will be introduced in Section 3.1. The strict-size logic itself is
3.1 Formal Concepts

In this section we introduce a set of formal concepts which are used in the strict-size logic.

First, some terminology needs to be clarified about Java/FJEU programs. A Java program consists of classes, which have methods (functions on a certain class) and fields (attributes). An instance of a class is called an object. A method is a function which is defined for a certain class. It is always called on an instance of this class and takes this object through the special variable this. A field contains a reference (pointer) to another object, or to null. Multiple variables may point to the same object on the heap, in which case these variables are called aliases.

3.1.1 Class Ontology

In Java, classes may extend another class, in which case the fields and methods of the superclass are inherited by the subclass. For the sake of simplicity, in FJEU, fields and methods are not allowed to be redefined in a subclass.

For example, consider the data structure for trees, as given in Listing 3.1. The field elem is defined in the class Tree. It is said to be inherited by the subclass Node, from its superclass Tree. The fields left and right are added to the class Node and are not a part of the superclass Tree. Methods may also be inherited from a superclass or added to a subclass.

The class ontology relation $D <: C$ is defined as class $D$ being a subclass of $C$. The class ontology relation is reflexive ($C <: C$), transitive (if $D <: C$ and $E <: D$ then $E <: C$) and antisymmetric (if $D <: C$ and $C <: D$ then $C = D$). It is therefore a partial order on classes. The class ontology relation defines a tree structure of classes, where the classes lower in the structure have more available fields and methods than their superclasses.

The class ontology relation is defined statically with respect to the strict-size logic.

3.1.2 Fields

We introduce the partial mapping $A$ from class and field names to class names, defining the types of fields:
3.1. FORMAL CONCEPTS

A : ClsName × FieldName → ClsName

For instance, A(List, next) = List. This mapping is used to store the defined types of fields. A similar mapping is introduced for methods in Section 3.1.6 but other formal concepts need to be defined first.

The mapping A(C, a) specifies the fields that are defined for the class C. For example, A(Node, left) = Tree, but A(Tree, left) is undefined. The fields that class C inherits from its superclasses are also included in A. We will refer to the collection of all fields that are accessible in C (i.e. that are part of the domain of A when combined with C) by Fields(C). Formally:

\[
\text{Fields}(C) := \{a \mid \exists D. \ (A(C, a) = D)\}
\]

For example, Fields(List) = \{next\}, Fields(Tree) = \{elem\} and, because of inheritance, Fields(Node) = \{elem, left, right\}. The field elem is defined in Tree and inherited by Node. Class Node also has a left and right subtree.

If A(C, a) = D, this means that for an instance of C, field a is either a null reference, or a reference to an instance of D or any of Ds subclasses. We therefore refer to D as the static type of field a. This is the type of field a as stated in the declaration of class C. The dynamic type of a can be D or any of its subclasses.

A similar mapping, B, is introduced to store whether or not a field a is counted in the calculation of the size of a class. A field a is only counted if the annotation /@count/@ is present in its definition. As an example, consider the Node class of Listing 3.1. The fields left and right are counted, but the size of the inherited field elem is ignored in the calculation of the size of a Node. The definition of size and a further explanation of the counting of fields is given in Section 3.1.4. Formally, B is defined as:

B : ClsName × FieldName → [0, 1]

The value of B(C, a) is one if and only if the field a is defined for C and the aforementioned annotation is present in its definition.

Because the sizes of non-counted fields are ignored they must be strictly separated from other objects for which the size may be important elsewhere, in order to avoid various problems. For example, when a non-counted field is updated, all sizes remain constant in the logic. If this non-counted field is however an alias of a counted field, the size of the counted field is corrupted. This separation is not enforced by the logic, i.e. it is left to the programmer.

3.1.3 Aliasing and Reachability

Accessing a field can be done in Java and FJEU by using the following syntax: x.a. An access path is a sequence of such field accesses, for instance x.next.next.next.
Formally, an access path is defined as $x.a_1 \ldots a_k$, where $x \in \text{Var}$, $k \geq 0$ and $a_i \in \text{Fields}(C_{i-1})$, where $C_{i-1}$ is the class of $x.a_1 \ldots a_{i-1}$.

Access paths that point to the same heap location (i.e., to the same object) are called aliases, denoted by $path_1 \sim path_2$. Formally:

1. The paths are syntactically or semantically equal
   
   (a) $path_1 \equiv path_2$, for instance $x \sim x$, $z.\text{next} \sim z.\text{next}$.
   
   (b) $path_1 = path_2$, for instance $x.\text{right} = y$, thus $x.\text{right} \sim y$, after the update $x.\text{right} \leftarrow y$.

2. $path_1 \sim path'_1.a_1 \ldots a_k \land path_2 \sim path'_2.a_1 \ldots a_k$, with $k \geq 0$, and $path'_1 \sim path'_2$. For instance, in Figure 3.1, $x.\text{right}.\text{left} \sim y.\text{left}$ because $x.\text{right} \sim y$.

From this definition it follows that the aliasing relation is reflexive ($x \sim x$), symmetric ($x \sim y \Rightarrow y \sim x$) and transitive (($x \sim y \land y \sim z) \Rightarrow x \sim z$).

![Figure 3.1: Example data structures (note that only the counted references are depicted)](image)

The reachability relation $x \rightarrow y$ means that there is an access path to $y$, from $x$ or one of its aliases:

$\text{path}_1 \rightarrow \text{path}_2$ iff $\text{path}_2 = \text{path}'_1.a_1 \ldots a_k$, where $k \geq 0$ and $\text{path}'_1 \sim \text{path}_1$.

For example, in Figure 3.1 $x \rightarrow y.\text{left}$ and $z \rightarrow z.\text{next}$.next, but $y \not\sim x$.

From this definition it follows that the reachability relation is reflexive and transitive, and that if $x \sim y$, then $x \rightarrow y$ and $y \rightarrow x$.

Because reachability can always be derived from aliasing relations and access paths, the relation is not explicitly stored to avoid redundancy.

Reachability shows some similarity to ownership types [16, 15, 11], which can be used for local reasoning about individual objects by enforcing a specification of their
encapsulation. However, while ownership of a certain attribute is a property of an object, reachability of all the attributes of an object is guaranteed by definition.

Circular data is not considered in this paper. Paths path_1 and path_2 are circular iff path_1 → path_2 ∧ path_2 → path_1 ∧ path_1 ≠ path_2. An example is shown in Figure 3.2. In this figure the predicate holds for any two objects from the set \{x.next, x.next.next, x.next.next.next\}. For instance, x.next → x.next.next ∧ x.next.next → x.next ∧ x.next ≠ x.next.next.

![Figure 3.2: Circular data](image)

The absence of circular data is ensured by the field update rule, which is given in Section 3.2.6.

Multiple access paths can lead from one object to another. The function paths gives the number of paths from an object \(x\) to an object \(y\) and is defined as:

\[
paths(x, y) = \begin{cases} 
0 & \text{if } x \not\rightarrow y \\
1 & \text{if } x \rightarrow y \\
\sum_{a \in \text{Fields}(C)} paths(x.a, y) & \text{else where } C \text{ is the class of } x
\end{cases}
\]

### 3.1.4 Sized Types

The size of an object is defined to be the sum of the sizes of all the counted fields, plus one. For instance, the size of the tree structure in Figure 3.1(a) is five and the list structure in Figure 3.1(b) is of size three. The mechanism to count or not count certain fields is used to be able to calculate the size of the considered data structure only, excluding the sizes of elements. For example, the size of a list should be its length. If all fields were counted the size of a list would be its length plus the sum of the sizes of all the elements of the list.

The size of an object is given by a size expression, which may incorporate size variables. The total set of size variables is denoted by \(SV\).

\[
\text{SizeExpr} := m \mid c \mid \text{SizeExpr} + \text{SizeExpr} \mid \text{SizeExpr} - \text{SizeExpr} \\
\mid \text{SizeExpr} \cdot \text{SizeExpr} \mid \text{max}_1(\text{SizeExpr})
\]
with \( m \in SV \) and \( c \in \mathbb{R}^+ \).

The relation \( \max_1(n) \) is defined as \( \max(n, 1) \). It can be used here because an object always has a minimal size of one, so for example a method can never decrease the size of this to \( n \) if \( n < 1 \).

The size of instances of a certain class is given by the sized-type specification of that class. By sized-type specifications we mean size expressions that, instead of size variables, range over the sizes of the fields of a class, for instance \( n_{\text{next}} + 1 \) for the length of lists, or \( n_{\text{left}} + n_{\text{right}} + 1 \) for the size of a node in a tree. Here, \( n_a \) refers to the size of the field \( a \). Formally:

\[
\text{SizedTypeSpec} := n \mid c \mid \text{SizedTypeSpec} + \text{SizedTypeSpec} \mid \max_1(\text{SizedTypeSpec} - \text{SizedTypeSpec}) \mid \text{SizedTypeSpec} \ast \text{SizedTypeSpec}
\]

As opposed to the size variables \( m \), \( n_{x,a} \) is a function which returns the current size of field \( a \) of an object \( x \). Therefore the value of \( n_{x,a} \) may change in time, while a size variable \( m \) (possibly decorated with a subscript) represents a certain unknown, but fixed, value.

The size of an instance \( x \) of a class is calculated by adding one to the sum of the sizes of its counted fields. The function \( \Delta \) generates the sized-type specification for an object \( x \) of class \( C \). Formally:

\[
\Delta(x, C) = 1 + \sum_{a \in \text{Fields}(C)} B(C, a) \cdot n_{x,a}
\]

For example, the size of a list that is null-terminated (as defined in Listing 2.4) is equal to its length. If the list is terminated by a separate \texttt{Nil} object, this object will also be counted, because the size is equal to the total amount of counted objects in a list structure. Another example of this is a tree structure as defined in Listing 3.1. Here, branches of the tree are terminated by \texttt{Tree} objects, which function as leaves. Therefore the size of the tree in Figure 3.1(a) is five.

### Dynamic Sized Types

Dynamic sized types are defined as a pair of a class name and a size expression, for instance (\texttt{List}, 5) for a list with a length of 5, or (\texttt{Tree}, \( n \)) for a tree of size \( n \).

Formally, a dynamic sized type is defined as:

\[
\text{DynamicSizedType} := (\text{ClsName}, \text{SizeExpr})
\]

A size expression evaluates to a non-negative integer. The statement \( x : (C, p) \) means that \( x \) is either an instance of class \( C \), when \( p > 0 \), or null, when \( p = 0 \).

If two access path are aliases they refer to the same object and must therefore have the same sized type:
Lemma 1 (Aliased sizes): $\text{path}_1 \sim \text{path}_2 \land \text{path}_1 : (C, p) \rightarrow \text{path}_2 : (C, p)$

Proof $\text{path}_1$ and $\text{path}_2$ point to the same location on the heap, thus to the same object. Therefore their sized types are equal.

Static Sized Types

There are cases where the dynamic type of a variable is not known at compile-time. For example, a parameter $x$ of a method $m$ may have the static type $\text{Tree}$, which is in the method definition and means that at runtime $x$ is an object of class $\text{Tree}$ or any of its subclasses, $\text{Node}$ for instance. We denote the static type of $x$ as $x : [\text{Tree}, p]$.

The collection $\text{SizedType}$ is defined as:

$$\text{SizedType} := \text{DynamicSizedType} | \text{StaticSizedType}$$

A program variable always refers to either null, in which case the size of the variable should be 0, or to an instance $x$ of a class $C$, in which case the size of the variable should be equal to $\Delta(x, C)$. This is captured in Definition 1:

**Definition 1** (Correctness of the typing): Typing $x : (C, p)$ is consistent iff:

- $x = \text{null} \rightarrow p = 0$
- $x \neq \text{null} \rightarrow p = \Delta(x, C)$
The size of a field \( a \) of object \( x \) is given by the special size variable \( n_{x,a} \). If known, the value of this variable may be calculated using the following definition:

**Definition 2** (Field size): \( x.a : (C, p) \rightarrow n_{x,a} = p \)

Because of the definition of size, the size of an object can always be expressed by a linear expression on the size of its fields. However, the annotated pre and postconditions of a method can be polynomial. For example, a method which takes a list of size \( n \) as a parameter may be annotated to have a list of size \( n \cdot n \) as result. This \( n \cdot n \) is however equal to \( 1 + n_{\text{result.next}} \) if the annotation is correct.

Even though data structures in which multiple access paths from one object to another exist can be handled by the strict-size logic, it will lead to overestimation of the size of the structure. An example of such a data structure is shown in Figure 3.3.

![Figure 3.3: Doubly referenced object](image)

The size of the data structure in Figure 3.3 is obviously two, but will be calculated as three, because the size of \( x.left \vartriangleleft x.right \) will be counted twice.

The size of an *actual object* is always greater than or equal to one. Because a method can only be invoked on an object, not on a null reference, whenever a variable \( \text{this} \) is used, its size has a minimum of one. This is formalized in Lemma 3.

**Lemma 3** (Minimal size of this): \( \text{this} : (C, p) \rightarrow p \geq 1 \)

**Proof** The variable \( \text{this} \) can never be a null reference. We can therefore derive from Definition 1 that \( p = \Delta(\text{this}, C) \). The value of \( \Delta(\text{this}, C) \) is defined as:

\[
\Delta(\text{this}, C) = 1 + \sum_{a \in \text{Fields}(C)} B(C, a) \cdot n_{\text{this},a}
\]

Because \( B(C, a) \) and \( n_a \) are non-negative integers for all \( C \) and \( a \), the value of \( p \) will always be greater than or equal to 1.

### 3.1.5 Conditions

The pre and postconditions for the strict-size logic are defined by the following grammar:

\[
\text{Cond} := \text{Var} : \text{SizedType} \mid \text{Path} \vartriangleleft \text{Path} \mid \text{Cond} \land \text{Cond}
\]
For instance, \( x: (D, n + m) \land y.a \rightarrow x \) is a condition.

The correctness of a condition depends on the correctness of the individual sized types and the equality in size and class between aliases:

**Definition 3** (Correctness of conditions): A condition is consistent iff:

- All typings in the condition are consistent according to Definition 1
- For all paths \( \text{path}_1 \) and \( \text{path}_2 \) that are aliases, i.e. \( \text{path}_1 \sim \text{path}_2 \) is part of the condition, if the dynamic sized types of both are known, i.e. \( \text{path}_1 : (C, p_1) \) and \( \text{path}_2 : (D, p_2) \) are part of the condition, then those sized types are equal: \( C = D \) and \( p_1 = p_2 \).
- For all paths \( \text{path}_1 \) and \( \text{path}_2 \) that are aliases, i.e. \( \text{path}_1 \sim \text{path}_2 \) is part of the condition, if the static sized types of both are known, i.e. \( \text{path}_1 : [C, p_1] \) and \( \text{path}_2 : [D, p_2] \) are part of the condition, then \( C <: D \) or \( D <: C \) and \( p_1 = p_2 \).
- For all paths \( \text{path}_1 \) and \( \text{path}_2 \) that are aliases, i.e. \( \text{path}_1 \sim \text{path}_2 \) is part of the condition, if the static sized type of \( \text{path}_1 \) and the dynamic sized type of \( \text{path}_2 \) are known, i.e. \( \text{path}_1 : [C, p_1] \) and \( \text{path}_2 : (D, p_2) \) are part of the condition, then \( D <: C \) and \( p_1 = p_2 \).

### 3.1.6 Methods

As in [25], \( M_{\text{body}}(C, m) \) maps class and method names to method bodies:

\[
M_{\text{body}} : \text{ClsName} \times \text{MethodName} \rightarrow e
\]

The signature \( \Sigma \) defines a mapping between pairs of class and method names and the pre and postconditions of the method. Formally:

\[
\Sigma : \text{ClsName} \times \text{MethodName} \rightarrow \text{Pre} \times \text{Post}
\]

The free program variables of an expression \( e \) are denoted by \( FV(e) \). By definition, \( this \in FV(M_{\text{body}}(C, m)) \). The free variables that occur in the pre and postconditions of a method relate to the free variables of the method body in the following way:

If \( \Sigma(C, m) = (P, Q) \), then \( FV(P) \subseteq FV(M_{\text{body}}(C, m)) \) and \( FV(Q) \subseteq FV(M_{\text{body}}(C, m)) \cup \{\text{result}\} \)

For the sake of convenience, if \( FV(M_{\text{body}}(C, m)) = \{\text{this}\} \cup \{z_1, \ldots, z_i\} \), we will denote the pair \( (C, m) \) as \( (C, m(z)) \). Always, \( z \) are exactly the parameters of the method \( m \).

For example, \( \Sigma([\text{List}, \text{append}(x)]) = (\{\text{this} : (\text{List}, n), x : (\text{List}, m}\}, \{\text{this} : (\text{List}, n + m)\}) \), where \( x \) is the single parameter of the \text{append} method. For
a function that removes the last element from a list, the following would be
added to the signature: $\Sigma\text{(List, stripLast)} = \{\text{this : (List, n)}, \{\text{result : (List, max}_1(n - 1))}\}$. When this method is executed on a list of length 1 the
last element cannot be removed (a method cannot remove the object on which it
executes). This is implicit in the $\text{max}_1(n - 1)$ operation, which ensures the size
will remain at a minimum of 1.

3.1.7 The $\Omega$ System of Equations

As described in Section 3.1.4, the sized type specification of a class ranges over
the functions $n_{x,a}$ representing the current size of field $a$ of object $x$. There are
however cases where the value of one of these functions at a particular time needs
to be stored. For this goal, the predicate $\text{old}()$ is introduced, which, combined
with an $n_{x,a}$, stores the size of field $a$ of object $x$ at the time it is introduced.
Thus, $\text{old}(n_{x,a})$ is a size variable.

These special size variables are stored outside of the conditions, because they are
not relevant for any of the logical rules, except for the conditional rule, described
in Section 3.2.8. They are only necessary at the end of the verification of a size
annotation, to be able to express the sizes before execution (i.e. the sizes in the
precondition) in terms of up-to-date program variables. When a new size variable
is introduced in the logic, a link is made between this new variable and the field
size it represents.

To store such links, the system of equations $\Omega$ is introduced. Formally:

$$\Omega = \emptyset \mid \text{SizeVar} = \text{SizeVar} \mid \text{SizeExp} = 0 \mid \text{SizeExp} \geq 1$$

Whenever a new variable $\text{old}(n_{x,a})$ is introduced, any occurrences of $\text{old}(n_{x,a})$ in
$\Omega$ are replaced by $\text{old}(\text{old}(n_{x,a}))$. At any time, the value of $n_{x,a}$, expressed in
the size variables used in the logic, at method entry (over which the precondition is
specified) can be retrieved by fetching the oldest version in $\Omega$.

Because $\Omega$ is relatively constant through the strict-size logic rules, it is notated as:

$$\{\text{Cond}\}_{\Omega}$$

Because it is not used frequently in the logic, it is mostly omitted.

The system of equations is also used to store some additional knowledge about
sizes, namely that when a certain branch of a conditional is taken, the size of the
variable on which value the branch is made is known to be either 0 or non-negative.
3.1.8 Fetching Field Sizes

When the sized type of a path is known, the sizes of its fields can be stored in a size variable. This can be very useful, because often the actual size of a field does not matter (or can be known), but the relation between that size and other factors (the size of a parameter for instance) is important.

The definitions, axioms and lemmas given so far hold at all times and can therefore always be used to derive new knowledge. However, the information that can be derived cannot be included in a condition (remember that conditions only contain sized types and aliasing information). The axiom given in this section does supply new information that can be included in a condition, namely it provides the size of a field in the form of a variable.

If the dynamic or static type of a path is known, the static sized type of its fields can be determined from mapping $A$. A fresh size variable $m$ is introduced as its size. The link of this variable with the $n$ function which value it represents must be retained and is therefore stored in $\Omega$.

**Axiom 2** (Field Size): $\text{path} : (C, p) \lor \text{path} : [C, p] \rightarrow \text{path}.a : [A(C, a), m]$

Where $m = \text{old}(n_{\text{path}, a})$ is added to $\Omega$. If $\Omega$ already contains occurrences of $\text{old}(n_{\text{path}, a})$, these are replaced by $\text{old}(\text{old}(n_{\text{path}, a}))$.

3.2 Axiomatic Semantics

In Java, a distinction can be made between *expressions* and *statements*. Expressions evaluate to a certain value, statements have only side effects. Unlike in functional languages, expressions may also have side effects. We denote the return value of an expression by the variable $\text{result}$.

In the strict-size logic $C, D, E, F$ will range over classnames, $x, y, z$ range over program variables, $p$ ranges over size expressions, $a$ denotes a field and $P, Q$ and $R$ range over conditions.

When we write $V[b / a]$ we mean that everywhere that $a$ occurs in $V$ it is substituted by $b$ ($a$ pushes $b$ away), where $V$ is (part of) a condition or size expression. For vectors $\vec{a}$ and $\vec{b}$ each element $a_i$ is substituted by the corresponding element $b_i$ in $V[\vec{b} / \vec{a}]$.

3.2.1 Basic Expressions

The rules for initializing new objects, null references and getting the value of a variable are relatively simple. The expression null outputs a null reference. Because
in our logic every sized type should be specified by a class as well as a size, the class type of \( result \) is initialized to a type variable. This type variable will be replaced by an actual class name by the let rule (see Section 3.2.2). The size of the result is set to zero.

\[
\{ \} \text{null \( (result : (c, 0)) \)} \quad \text{Null}
\]

When a new object is constructed, all its field are initialed as references to null, so the sizes of all fields are 0. Therefore the size of the new object is 1.

\[
\{ \} \text{new \( C \{result : (C, 1)\} \)} \quad \text{New}
\]

When the value of a variable is retrieved, the sized type of this variable is copied to \( result \). This rule should only be applied when the value of the variable is read, not when it is written to.

\[
\{x : (C, p)\} \times \begin{cases} 
  x : (C, p) \\
  \text{result} \leftarrow x 
\end{cases} \quad \text{Var}
\]

These constructs are expressions, not statements, because they evaluate to an output value.

### 3.2.2 Let-Binding

In a let-binding, let \( x = e_1 \) in \( e_2 \), the value that \( e_1 \) evaluates to is bound to the new variable \( x \). Therefore, the size of the variable \( result \) in the postcondition of \( e_1 \) becomes the size of \( x \) in the precondition of \( e_2 \). Also, if \( result \) is part of any aliasing expressions in the postcondition of \( e_1 \), it has to be replaced by \( x \) in the precondition of \( e_2 \).

For example, when a newly constructed object is bound, let \( \text{List} \ x = \text{new} \text{List} \) in \( e_2 \), according to the logical rule for new as specified in the previous section, the result of the constructor, which is the postcondition of \( e_1 \), is \( \{result : (\text{List}, 1)\} \). This will be changed to \( \{x : (\text{List}, 1)\} \) in the precondition of \( e_2 \).

Here, \( e_1 \) may only be an expression, not a statement, because a statement does not evaluate to a certain output value. In other words, it may only be a basic expression, method invocation, field access, field update or type cast.
The class $D$ of $result$ must be a subclass of $C$, but $result$ is not typecast to this class.

### 3.2.3 Sequencing

In the special let construct where the wildcard $\_\_\_$ is used instead of a variable name, the result of $e_1$ is discarded. In this case, the expression $e_1$ is only evaluated for its side effects. In this case it does not have to be validated that $D \subset C$. If a method does not have a return type, then its result must be discarded.

This partial case of the let rule is semantically equal to the sequencing rule. This rule is added to allow a conditional to be followed by another statement or expression (possibly another conditional). It cannot be applied correctly in other situations.

\[
\begin{array}{c}
\{P\} e_1 \{Q\} e_2 \{R\} \\
\{P\} e_1 e_2 \{R\}
\end{array}
\]

**Sequence**

### 3.2.4 Method Invocation

As described in Section 3.1.4, the $\Sigma$ function maps class and method names to pre and postconditions, as they are defined in the annotation of the method. This is the precondition, modulo the substitution of formal for actual parameters, that should hold before the method is invoked and the postcondition, modulo the substitution of formal for actual parameters, that holds after the invocation.

Method $m$ should be defined in class $C$ or any of its superclasses. For convenience we do not allow methods (or fields) to be redefined by subclasses.

\[
\Sigma(C, m(\overline{x})) = (P, Q)
\]

\[
\begin{array}{c}
x : (C, p) \\
P[\overline{y} / \overline{z}, x / this]
\end{array}
\]

\[
\begin{array}{c}
x.m(\overline{y}) \{Q[\overline{y} / \overline{z}, x / this]\}
\end{array}
\]

As $P$ and $Q$ are the pre and postconditions as specified in the size annotation of the method, they consist only of the sized types of the parameters (including
this). It is assumed (left to the programmer) that in a method call all parameters (including this) are separated, i.e. not reachable from one another.

3.2.5 Field Access

The field access rule simply makes the result variable an alias of x.a. This is all the information that is necessary, because the sized type of result can be derived from this. If x.a has an alias y of which the dynamic or static sized type is known this can be used to derive the dynamic or static sized type of x.a by applying Lemma 1 or Lemma 2 respectively. If such an alias is not available then the static sized type of x.a may still be derived from the dynamic or static sized type of x using Axiom 2.

\[
\frac{a \in \text{Fields}(C)}{\{\} x.a \{\text{result} \leftarrow x.a\} \text{ Access}}
\]

3.2.6 Field Update

To do an update on a field of an object (also referred to as an assignment), the type of the field is retrieved from mapping A to ensure that the assigned object y is of a subclass of A(C, a).

Field update is an expression which, different from assignment in Java, evaluates to the updated version of x.

There are two field update rules. The first one is for the simple case where the size of field a is not counted in the calculation of the size of x. In this case, updating the field does not affect the size of any object.

If x.a has any aliases before the update, these are discarded in the postcondition, because x.a will point to a new heap location.

To avoid the creation of circular data, x may not be reachable from y.

\[
\begin{align*}
D &: E & A(C, a) &= E & B(C, a) &= 0 & \text{Update I} \\
\{ x : (C, p_x) \} & & & & & & \\
y : (D, p_y) & & & & & & \\
x.a \leftarrow z_1 & & x.a \leftarrow y & & x : (C, p_x) & & \\
\vdots & & y : (D, p_y) & & & & \\
x.a \leftarrow z_k & & x.a \leftarrow y & & & & \\
y \not\Rightarrow x & & \text{result} \leftarrow x & & & & 
\end{align*}
\]
Note that, unless \( z_i \sim y \), all aliasing relations between \( x.a \) and \( z_1 \ldots z_k \) are broken.

When the size of field \( a \) is counted in the calculation of the size of \( x \) the sizes of \( x \) and its aliases must be correctly adapted. The new size of \( x \) is its old size minus the size of the old \( x.a \), plus the size of \( y \). Because the old sized type of \( x.a \) will be lost in the update, the sized type of one of its aliases must be saved, otherwise this information will be lost.

\[
\begin{align*}
E <: F \\
A(C, a) &= F \\
B(C, a) &= 1
\end{align*}
\]

Update II

\[
\begin{aligned}
x &\colon (C, p_x) \\
x.a &\colon (D, p_a) \\
y &\colon (E, p_y) \\
x.a &\sim z_1 \\
&\vdots \\
x.a &\sim z_k \\
z_i &\colon (D, p_{a_i}) \\
y &\sim x
\end{aligned}
\]

Note that the value of \( n_{x,a} \) does not need to be adapted, because it is not part of the conditions, but is calculated using Definition 2. As in the Update I rule, all aliasing relations between \( x.a \) and \( z_1 \ldots z_k \) are broken.

### 3.2.7 Casting

In FJEU, only upcasting is allowed, which means that instances of a certain class may only be cast to one of their superclasses. This cast expression outputs an object of the requested superclass, from which all attributes defined not by this superclass or its own superclasses, but by its subclasses, are removed. Therefore, the corresponding sizes must also be subtracted from the size of \( y \).
3.2.8 Conditionals

Since there are no boolean expressions in FJEU, conditionals check whether or not an object is of a certain class (or one of its subclasses).

\[
\begin{align*}
\{ & \text{C <: D} \} \\
& \{ x : (C, p) \}_{\Omega \cup \{p \geq 1\}} \\
\} & e_1 \{ Q_1 \}_{\Omega_1} \quad Q_1 \vdash Q
\end{align*}
\]

\[
\begin{align*}
\{ & \text{C ≠ D} \} \\
& \{ x : (C, p) \}_{\Omega \cup \{p \geq 1\}} \\
\} & e_2 \{ Q_2 \}_{\Omega_2} \quad Q_2 \vdash Q
\end{align*}
\]

\[
\begin{align*}
\{ & x : (C, p) \}_{\Omega \cup \{p = 0\}} \\
\} & e_3 \{ Q_3 \}_{\Omega_3} \quad Q_3 \vdash Q
\end{align*}
\]

\[
\{ x : (C, p) \}_{\Omega_{1,2}} \quad \text{if } x \text{ instanceof } D \text{ then } e_1 \text{ else } e_2 \{ Q \}_{\Omega}
\]

Note that the postcondition \( Q \) of the conditional should be derivable from the postconditions of all branches. This means that if \( Q_1 \vdash x.a \sim y \), but \( Q_2 \nvdash x.a \sim y \), \( x.a \sim y \) will not be a part of \( Q \). \( Q \) must be chosen such that there is no \( Q_i \) for which \( Q_1 \vdash Q_i, Q_2 \vdash Q_i, Q_3 \vdash Q_i \) and \( Q \nvdash Q_i \). In other words, \( Q \) must be the strongest condition that is derivable from any of \( Q_1, Q_2, \) or \( Q_3 \).

For sizes this means that methods in which different branches yield different sizes (that cannot be rewritten into one common expression) can not be handled by the logic (they are considered to be outside of the scope of this work). The exact limitations are discussed in Section 6.1.1 along with hints to a possible solution.

The implications of this construction for aliasing are discussed in Section 3.3, where it is explained how the incompleteness of the aliasing information in \( Q \) can be handled.

The \( \Omega \) of the precondition is restored in the postcondition. If any new equalities have been added to it in one of the branches, these can not be in \( Q \), because \( Q \) must be derivable from all branches.

3.2.9 Return Statements

Return statements are just in methods to end their execution and return the result value of the method. Depending on the return type of the method, there are two possible forms of the statement.

When the return type is a certain class and the method is correctly typed, then \( e \) should have a \textit{result} variable of this class in its postcondition. This \textit{result} is a part of \( Q \), so it automatically becomes the result of the entire return statement.
3.2. AXIOMATIC SEMANTICS

\[
\begin{align*}
\{P\} & \text{ e } \{Q\} \\
\{P\} & \text{ return e } \{Q\}
\end{align*}
\]

Return I

When the return type of a method is `void`, then return just ends the interpretation of (a certain branch in) the method.

\[
\{\} & \text{ return } \{
\}
\]

Return II

When we check the correctness of a method annotation, we verify that it holds at every return statement, because that is the end of the method (for a certain branch).

3.2.10 Frame Rule

Local reasoning \[30, 37, 40\] is a technique to reason about separate parts of the heap in a Hoare logic. This makes it possible to omit unrelated information from conditions, simplifying the production rules and greatly improving the scalability of the logic.

Intuitively, if there is a part of the precondition \(R\) that is independent from the rest of the precondition \(P\), it should be possible to put aside this \(R\) for a while if it is not modified by \(e\). This is what the frame rule does.

The standard frame rule is given as:

\[
\begin{align*}
\{P\} & \text{ e } \{Q\} \\
\{P + R\} & \text{ e } \{Q + R\}
\end{align*}
\]

StandardFrame

Where \(P + R\) means that the heap can be split into two disjoint parts such that \(P\) holds for one part and \(R\) holds for the other. In our terms this means that variables in \(P\) may not be reachable from \(R\) and vice versa. However, in the strict-size logic it should also be possible to reason about \(P\), disregarding \(R\), when variables in \(P\) can be reached from \(R\). We therefore introduce a less restrictive version of the standard frame rule, applying two modifications.

First, \(R\) may contain variables from which variables in \(P\) can be reached. However, the size of these variables depends on the size of the variables in \(P\) and must therefore be correctly adapted. The condition \(R’\) is introduced for this purpose, which is constructed from \(R\) by updating these sizes.

Second, a new side condition is needed. The standard frame rule has the side condition \(\text{modifies}(e) \cap \text{FV}(R) = \emptyset\), where \(\text{modifies}(x.a \leftarrow b)\) is usually defined
to be the empty set (for example in [38]), expressing the independence of \( x \) on the values of its fields (here \( a \)). However, for our sized types, the size of \( x \) does depend on (the sizes of) its fields. For instance, the type of a \( \text{List} \) object \( x \) is independent of the type of the object that \( x.\text{next} \) refers to for regular types, but the size of \( x \) is given by the length of \( x.\text{next} \) plus one.

The side condition must prevent that any of the free variables in \( R \) is changed by \( e \), therefore it is changed to:

\[
\forall x \in \text{FV}(R). \forall y \in \text{modifies}(e) \cdot y \not\in x
\]

The set \( \text{modifies}(e) \) is defined as follows:

- \( \text{modifies}(x.a \leftarrow y) = \{ x.a \} \) if \( a \) is counted, \( \emptyset \) otherwise
- \( \text{modifies}(\text{let } C \ x = e_1 \text{ in } e_2) = \text{modifies}(e_1) \cup \text{modifies}(e_2) \)
- \( \text{modifies}(\text{if } x \text{ instanceof } D \text{ then } e_1 \text{ else } e_2) = \text{modifies}(e_1) \cup \text{modifies}(e_2) \)
- \( \text{modifies}(x.m(p)) = V[x / \text{this}, p / \bar{a}] \)

Where \( V = \{ v \mid v \in \{ \text{this}, \bar{a} \} \land v \rightsquigarrow w \land w \in \text{modifies}(M_{\text{body}}(C, m(\bar{a}))) \)\), where \( x \) is an instance of \( C \)

\( \text{modifies}(\_.) = \emptyset \)

The rule for a method invocation needs some more explanation. The sizes that may be modified by a method call that are of interest are those of its parameters (including the object on which the method is invoked). Variables that are modified but that are local to the method body are irrelevant. If a field (or something further away) of a parameter is changed, it is not relevant which field exactly, just that the size of that parameter is modified. Therefore, the set of variables that are modified by a method call is given by the set of parameters of that method, from which the variables that are changed in the method body can be reached.

Note that the side condition is not explicitly stated in the rule and that the sizes of free variables in \( R \) may still be changed by \( e \). The frame rule for the strict-size logic is specified as follows:

\[
\frac{\{ P \} e \{ Q \}}{\{ P \land R \} e \{ Q \land R' \}} \quad \text{Frame}
\]

Where \( R' \) is constructed from \( R \) in the following way:

For all \( x : (C_x, p_x) \in R \), for which there exist \( y : (C_y, p_y, \text{old}) \in P \) and \( y : (C_y, p_y, \text{new}) \in Q \), with \( P \land R \models x \rightsquigarrow y \), add the following to the post-condition \( R' \) (and omit the sized type of \( x \) as specified in \( R \)):

\( x : (C_x, p_x - p_y, \text{old} \cdot \text{paths}(x, y) + p_y, \text{new} \cdot \text{paths}(x, y)) \).
3.2. AXIOMATIC SEMANTICS

All aliasing relations in $R$ are copied unchanged to $R'$. 

![Diagram](image_url)

**Figure 3.4:** Visualization of a field update $y.a \leftarrow z'$.

The size of $R$ should always be maximized (and therefore the size of $P$ is minimized). In other words, if a sized type or an aliasing relation can be a part of $R$, then it should be a part of $R$.

An example of the use of the frame rule in conjunction with the Update II rule is visualized in Figure 3.4. The set $\text{modifies}(y.a \leftarrow z')$ here consists of $y.a$ and $z$ (because it is an alias of $y.a$). The frame rule is used to reason locally about $z$ and $z'$ (and the objects that can be reached from those), $P$, and ignore the rest of the objects, $R$. In fact, $y$ could have also been a part of $R$ instead of $P$, but the frame rule is used here with the field update rule, which requires the sized type of $y$ in its precondition.

In a method invocation, if the size of a certain variable is changed, this may affect the sizes of other related variables as well. If for instance the size of $x$.next is changed, the size of $x$ needs to be changed accordingly. Because the frame rule supplies this functionality and to avoid redundancy by adding elements of the frame rule to the invoke rule, the method invocation rule may only be used in conjunction with the frame rule.

The same restriction applies to the Update II rule, for the same reasons. If the size of a field of a variable $x$ is changed, the sizes of all variables $y_1 \ldots y_n$, for which $y_i \rightarrow x$ holds need to be adapted accordingly.

A conjunction of the frame rule with the Invoke rule or the Update II rule should be regarded as a single rule in the sense that the latter two rules may never be applied without first applying the frame rule, the frame rule should never be applied without either the Invoke rule or the Update II rule (although it could, but for the future goal of an implementation we should be exact), and that Axiom 2 may not be applied above the line of a frame rule. The reason the frame rule is given as a separate rule here is to avoid redundancy between the method invocation and...
field update rules and to stress the similarity with separation logic.

The conjunction of the frame rule with the Invoke or Update II rule is easily transformed into a single rule by respectively replacing \( P \) and \( Q \) with the pre and postcondition of the rule. For completeness (because they are used as a single rule in practise) these are listed here.

For both rules, the side condition of the frame rule must hold and \( R' \) is constructed from \( R \) as described above.

### 3.3 Alias Control

Changes in the aliasing between two objects can occur in either of two ways: a field update or a method invocation (let just introduces a new variable). If this occurs within a conditional (i.e. the aliasing relation is changed conditionally), then this may lead to problems. Consider the example in Listing 3.2.

As stated in Section 3.2.8 the postcondition of a conditional must be derivable from the postconditions of all branches. Because \( \text{this.next} \rightarrow x \) will only be true after the then-branch, not after the else-branch, it is not derivable from both postconditions and not part of the postcondition of the conditional. This information is therefore lost. We say that after such a conditional the aliasing information is incomplete.
3.3. ALIAS CONTROL

Listing 3.2: Conditional aliasing

```plaintext
if x instanceof List then
  let List _ = this.next <- x in
else
  let List _ = x in null
let List _ = x.next <- y in
return
```

Such incomplete aliasing information may be hazardous for the strict-size logic. If, for example, the conjunction of the field update rule and the frame rule is applied to the field update on line 5 of Listing 3.2 without the information that this.next ← x, then the size of this will not be correctly adapted (it will remain constant).

We therefore want to rule out such examples. But simply stating that no aliasing may be changed within a conditional or that after a conditional the frame rule may not be applied anymore would be a too harsh restriction. We therefore look for cases in which incomplete aliasing information is harmless to the logic, in an attempt to maximize the set of problems that it can handle.

3.3.1 Harmless Conditional Aliasing

To be able to describe the cases in which conditional aliasing is harmless we first need the define the notion of scope. Then three cases in which conditional aliasing is harmless are described.

Scope of Variables

The scope of a newly constructed object is the innermost branch in which the object is created (we say that the variable is local to that branch). This may be the method itself if the object is created outside of any conditional. A newly constructed object may be one that is created with a new expression or one that is returned by a method and does not have any aliasing relation to other objects. This is the case if there is no mayaffectaliasing annotation between ∩ result and another object for this method. By this definition, the scope of the parameters of a method (including this) can never be local.

An example of a variable that is local to a certain branch is shown in Listing 3.3. The scope of n is limited to the innermost then-branch (lines 3 and 4), a use of n outside of its scope would yield a compiler error.
A variable escapes its scope if it is returned by the method. So, variables that are returned at the end of a branch may not be considered local to that branch. It may be the case that a variable with a wider scope is returned in one branch (with a smaller scope), but not in another. In that case the variable may not be considered local in the branch where it is returned or before entering that branch (because it might be returned then), but it may be considered a local variable in the branch where it is not returned. An example of this is described in Section 4.3.

Furthermore, if a local variable is assigned to the field of a variable that is not local, then it may not be considered local afterward, because it has a non-local alias so it will escape the branch.

### Aliasing of Local Variables

In the example shown in Listing 3.3, the scope of n is limited to the innermost then-branch. The fact that it is appended to y changes the size of y and the aliasing relation between y and n. But, of course, this change in aliasing is irrelevant outside of the scope of n.

In general, when at least one of the variables for which the aliasing relation is affected is local to the current innermost branch, the change in aliasing is harmless. Note that if the aliasing between a local and a non-local variable is affected, the former may not be considered local anymore afterward, because it now has a non-local alias.

It can be read from the mayaffectaliasing annotations of a method whether it may affect the aliasing relation between certain variables (and its result). These variables are given in pairs, of which at least one element must be local in order for a method invocation to be harmless.

Listing 3.3: The scope of n is restricted to the innermost then-branch

```java
if x instanceof List then
  if y instanceof List then
    let n = new List in
    y.append(n);
  else
    ...
else
  ...
```
3.3. ALIAS CONTROL

End of the Method

The second case where conditionally changing aliasing information is harmless to the logic is when the information is changed at the end of the method. This is the case in Listing 3.4.

```plaintext
if x instanceof List then
  let List _ = this.next <- x in
  return
else
  ...
```

Listing 3.4: Conditional aliasing at the end of the method

While the aliasing relation between this.next and y is conditionally changed on line 2, this aliasing relation is not important anymore afterward, because the method has then returned. It is not important whether or not this and y are local in this case.

Non-Counted Fields

The final case where conditional aliasing is harmless is simply if the aliasing of a field which is not counted. If such a field is changed this cannot affect the size of the object itself and is therefore harmless.

If however this field would contain an object for which the size is important elsewhere (for example a list which length is also determined by the logic) then this could lead to problems. The same holds for cases in which the contents of a field that is not counted are copied to a field that is counted. It is left to the programmer to keep a strict separation between the fields that are counted and fields that are not, in order to avoid these situations. This is not enforced by our logic.

3.3.2 Verifying That All Aliasing is Harmless

The verification that all conditional aliasing in a certain method is harmless consists of two parts. First we need to validate the mayaffectaliasing annotation of all the methods that are invoked from this method. Then, we need to make sure that whenever aliasing information is changed conditionally, at least one of the three cases described in Section 3.3.1 applies.
CHAPTER 3. PROGRAM LOGIC FOR STRICT-SIZE ANALYSIS

Checking Method Annotations

To verify the mayaffectaliasing annotation of a method we must ensure that no aliasing is changed between the parameters of the method (including its result and this), unless it is explicit in the annotation. Because there is no separate annotation for the case where the aliasing between two parameters is surely changed, this also applies to changes outside of conditionals.

We are looking for field updates and method invocations in which pairs of non-local variables are affected. This may occur anywhere outside a conditional or inside a conditional if it is immediately followed by a return. If a recursive call to the method under investigation is present then the annotation must temporarily be assumed correct.

When an occurrence of a field update or method invocation where two non-local variables are affected is found, a mayaffectaliasing annotation must always be present for each pair of parameters from which these variables can be reached.

If the aliasing between two parameters $a$ and $b$ is changed and then the aliasing between one of those parameters (say $a$) and another parameter ($c$) changes, then there is the possibility that the aliasing between $b$ and $c$ is also affected. Therefore, when a mayaffectaliasing pair is added and one (or both) of the parameters that make up the pair are already part of such relationships, then pairs must be added for all other parameters that are part of those relationships.

Ensuring No Use of Misinformation

After we have validated the aliasing annotations of all the invoked methods we must convince ourselves that no incomplete aliasing information can be used anywhere in the method.

We are looking for locations where aliasing is changed conditionally between two variables that are not local to the innermost branch in which they are changed. If such a location is found and is not immediately followed by a return (the second case of Section 3.3.1) then this may lead to incorrect aliasing information, so the method cannot be validated by the logic.
Chapter 4

Examples of Annotation Verification Using the Strict-Size Logic

The strict-size logic will be illustrated with a series of examples in this section. What we need to prove is that the size annotations for the given methods are correct. The general way to do this is to assume that the annotation precondition holds and derive the postcondition from it by applying the strict-size logic to the method body. The postcondition should be derivable at the points where the method may end, which in general are after a return statement, or sometimes simply the end of body.

Before we can reliably employ the strict-size logic, however, we have to convince ourselves that no incomplete aliasing information is used anywhere in the method, as is described in Section 3.3.2. To do this we have to verify that the mayaffectaliasing annotations of all the methods that are invoked are correct. This may include the considered method itself, in the case of a recursive call.

For readability, only single logical rules are shown in the examples. If a derivation extends after a rule (for instance, in the let-rule), this derivation is listed as a variable $En$ and worked out later.

4.1 List Concatenation

As a first example we will consider the append method, which concatenates two lists. The implementation of the List class and the append method is the same as in Listing 2.4. For readability it is repeated in Listing 4.1.
The size of a list is equal to its length: one plus the size of its tail. When two lists are appended, the size of their concatenation is equal to the sum of the sizes of both lists. This is expressed in the annotation of the `append` method. The List object on which the method is invoked is appended by the list that parameter `l` refers to and returned.

```java
class List {
    Object elem;
    /*@ count @*/ List next;

    /*@ 
    @ requires (\sizedtype l; List{pl})
    @     && (\sizedtype \this; List{pt})
    @ 
    @ ensures (\sizedtype l; List{pl})
    @     && (\sizedtype \this; List{pl+pt})
    @ 
    @ mayaffectaliasing \this; l
    @*/
    void append(List l) {
        let List n = this.next in
        if n instanceof List then
            let List _ = n.append(l) in
            return
        else
            let List _ = this.next <- l in
            return;
    }
}
```

Listing 4.1: The append() method in FJEU

**Verifying the Aliasing Annotations**  We have to start by verifying that the `mayaffectaliasing` annotations of all the invoked methods are correct. In this case the only method invocation is a recursive call to `append` itself.

As is described in Section 3.3.2 we are looking for field updates and method invocations where the aliasing relation between pairs of non-local variables is affected. This occurs in two locations in the `append` method.

The first is the recursive invocation of `append` on line 17. The two non-local variables affected here are `n` and `l`. It is easy to see that `n` is an alias of `this.next` and therefore the affected parameters of this method invocation are `this` and `l` (this `next` can be reached by `this` and no other parameters), matching the annotation.
The second location where the aliasing relation between two non-local variables is changed is the field update on line 20. Again, the affected parameters are this and l, matching the annotation.

So, the pair of parameters this and l is the one and only pair between which aliasing may be affected by invoking the append method (in fact it is the only possible pair, since the method does not have a result). This means the mayaffectaliasing annotation is correct.

Ensuring No Misinformation is Used  We now have to convince ourselves that no incomplete information regarding aliasing can be used anywhere in the logic, by following the instructions in Section 3.3.2. The two critical points in the method where aliasing may be affected are the recursive call of append and the field update, as discussed in the previous paragraph. Both would not normally be considered safe to the logic, because they affect a pair of variables not local to their respective branches (this.next and l). However, both fall under the exception described in the end of Section 3.3.2 and are therefore harmless to the strict-size logic.

Verifying the Size Annotations  The annotation of the append method is translated to the precondition \( l : [\text{List}, p1] \land \text{this} : [\text{List}, pt] \) and the postcondition \( l : [\text{List}, pl] \land \text{this} : [\text{List}, pt + pl] \).

Because List has no subclasses the static sized types of the precondition can be easily transformed to dynamic sized types using Axiom 1.

\[
\begin{align*}
\text{this} : [\text{List}, pt] & \rightarrow \text{this} : (\text{List}, pt) \\
\text{l} : [\text{List}, pl] & \rightarrow \text{l} : (\text{List}, pl)
\end{align*}
\]

The signature of the append method as stated in the annotation should hold for the method body with respect to the strict-size logic. To prove this, we assume the precondition and derive the postcondition from it using the strict-size logic.

\[
\begin{align*}
c_1 < : \text{List} & \quad E_1 & \quad E_2 \\
\left\{ \begin{array}{c}
l : (\text{List}, pl) \\
\text{this} : (\text{List}, pt)
\end{array} \right\} & \text{let List n = this.next in } \ldots \{Q_1\}
\end{align*}
\]

The first rule that we apply is the let rule. The postcondition \( Q_1 \) is equal to the postcondition of \( E_2 \) and is therefore not yet known at this point. Note that it is also not important, because we are proving that the postcondition holds at the return statements. Execution of the method can never get past the conditional. The class type \( c_1 \) of the output of this.next should be a subclass of List, but this is known only after completing derivation \( E_1 \).

A parameter may be of size 0 if it is a null reference. The size of this is always at least one, otherwise the method cannot be invoked (Lemma 3).
The proof \( E_1 \) consists of the access rule:

\[
\begin{align*}
\text{next} & \in \text{Fields}(\text{List}) \\
\{ & \text{l} : (\text{List}, pl) \\ & \text{this} : (\text{List}, pt) \} \; \text{this.next} \\
\{ & \text{l} : (\text{List}, pl) \\ & \text{this} : (\text{List}, pt) \}
\end{align*}
\]

Access

If we now look at the let rule, we see that we need the dynamic type of \( \text{result} \). However, it is clear that we can only know its static type. We have to do our proof for every possible dynamic type, which in this case is just \( \text{List} \).

This dynamic type can be derived by successively applying Axiom 2, Lemma 2 and Axiom 1:

\[
\begin{align*}
\text{this} : (\text{List}, pt) & \rightarrow \text{this}.next : [\text{List}, m_1] \\
\text{this}.next & \sim \text{result} \land \text{this}.next : [\text{List}, m_1] \rightarrow \text{result} : [\text{List}, m_1] \\
\text{result} & : [\text{List}, m_1] \rightarrow \text{result} : (\text{List}, m_1)
\end{align*}
\]

In the application of Axiom 2, \( m_1 = old(n_{\text{this.next}}) \) is added to \( \Omega \).

From the type of \( \text{result} \) it is now clear that \( c_1 \) is \( \text{List} \) and therefore that \( c_1 \prec \text{List} \) (in the let rule above) holds.

We continue with \( E_2 \) now, for which \( \text{result} \) has been substituted by \( n \) as specified in the let rule. When aliases exist in a condition (here \( n \) and \( \text{this.next} \)), the sized type of only one of these aliases is kept (here \( n \)), because the sized types of all aliases are equal. Otherwise, if a sized type would be changed, care should be taken to also change the sized types of all aliases. Now, the sized types of the aliases automatically change with the one that is stored.

\[
\begin{align*}
E_3 & \quad E_4 \\
\{ & \text{l} : (\text{List}, pl) \\ & \text{this} : (\text{List}, pt) \\ & n : (\text{List}, m_1) \\ & n \sim \text{this.next} \} & \text{if} & n \text{ instanceof List then } \ldots \text{ else } \ldots \{Q_1\}_\Omega
\end{align*}
\]

There are three rules above the line in the definition of the if rule, but because the class of \( n \) is \( \text{List} \), the precondition of the second rule can never become true.

The case where \( n \) is an instance of \( \text{List} \) is handled in \( E_3 \). We will continue with \( E_4 \) (the case where \( n \) is \( \text{null} \)) after we have completed \( E_3 \). In \( E_3, m_1 \geq 1 \) is added to \( \Omega \).
4.1. LIST CONCATENATION

The if-branch consists of a let-binding and a return. In the let-binding, \( E5 \) handles the recursive call to \( \text{append} \). In verifying this recursive invocation it is assumed that the \textbf{pre} and postconditions from the annotation hold. It does not have to be verified if the \textbf{result} of the method invocation is indeed a \textbf{List}, because it is discarded (in fact, it has no \textbf{result}).

Remember that the method invocation rule may not be applied without the frame rule. We therefore use the InvFrame rule here. For the method invocation we need the sized types of \( n \) and \( l \), so those are part of the P segment in the precondition to the frame rule. We can put the rest of the precondition in R, if the sidecondition of the frame rule \((\forall x \in \text{FV}(R). \forall y \in \text{modifies}(e). y \not\sim x)\) then holds.

To calculate the set \( \text{modifies}(n.\text{append}(l)) \), we need to determine what the contents of \( \text{modifies}(M_{body}(\text{List}, \text{append})(l)) \) is.

This set can be calculated step-by-step, starting with the entire method, which is a let-statement:

\[
\begin{align*}
\text{modifies}(\text{let List } n = e_1 \text{ in } e_2) &= \text{modifies}(e_1) \cup \text{modifies}(e_2) \\
\text{modifies}(e_1) &= \text{modifies}(\text{this.next}) = \emptyset \\
\text{modifies}(e_2) &= \text{modifies}(\text{if } n \text{ instanceof List then } e_3 \text{ else } e_4) \\
&= \text{modifies}(e_3) \cup \text{modifies}(e_4) \\
\text{modifies}(e_3) &= \text{modifies}(\text{let List } _ = e_5 \text{ in } e_6) = \text{modifies}(e_5) \cup \text{modifies}(e_6) \\
\text{modifies}(e_5) &= \text{modifies}(M_{\text{body}}(\text{List}, \text{append}(l)))
\end{align*}
\]

Since this is the set we are calculating we will ignore this recursion, by filling in \( \emptyset \) for \( \text{modifies}(e_3) \). No other parameters will be modified by a recursive call.

\[
\begin{align*}
\text{modifies}(e_6) &= \text{modifies}(\text{return}) = \emptyset \\
\text{modifies}(e_4) &= \text{modifies}(\text{let List } _ = e_7 \text{ in } e_8) = \text{modifies}(e_7) \cup \text{modifies}(e_8) \\
\text{modifies}(e_7) &= \text{modifies}(\text{this.next } \leftarrow l) = \{\text{this.next}\} \\
\text{modifies}(e_8) &= \text{modifies}(\text{return}) = \emptyset
\end{align*}
\]

So:

\[\text{modifies}(M_{\text{body}}(\text{List}, \text{append}(l))) = \{\text{this.next}\}\]

The set of the parameters of \( \text{append} \) from which \text{this.next} can be reached consists of \text{this}, which maps onto \( n \). So, \( \text{modifies}(n.\text{append}(l)) = \{n\} \).

We can now see that the side condition for the frame rule only holds if we also
put \( n \prec\) this.next in \( P \). The sized type of this is left in \( R \), because \( n \not\prec\) this.

As can be seen in the InvFrame rule, the variables in the annotation have to be substituted by the appropriate variables from the strict-size logic. In this case, the \( l \) in the annotation is the same \( l \) as in the logic. The variable \( n \) from the annotation should however be substituted by \( n \), because the method is called on \( n \), not on the this variable from the logic. Therefore, also the size \( pt \) in the annotation should be replaced by the size of \( n \): \( m_1 \).

In the annotation postcondition we see that the size of this becomes \( pt + pl \). This means that in the logic, the size of \( n \) becomes \( m_1 + pl \).

While the variables in the annotation apply to the method definition, they are here substituted by the variables in the method invocation.

\[
\Sigma(\text{List, append}(l)) = \left\{ \begin{array}{l}
1 : (\text{List}, pl) \\
\text{this} : (\text{List}, pt)
\end{array} \right\} + \left\{ \begin{array}{l}
1 : (\text{List}, pl) \\
\text{this} : (\text{List}, pt + pl)
\end{array} \right\}
\]

The dashed lines indicate the separation between \( P \) and \( R \) in the precondition and \( Q \) and \( R' \) in the postcondition.

We have constructed \( R' \) from \( R \) as described in Section 3.2.10. The sized type of this is a part of \( R \) and there exists \( n \) for which \( n : (\text{List}, m_1) \in P \) and \( n : (\text{List}, m_1 + pl) \in Q \), and this \( \prec n \). Therefore we must replace the sized type of this as given in \( R \) by its new sized type: \( \text{this} : (\text{List}, pt - m_1 + m_1 + pl) \). The new size of this is thus calculated by subtracting the old size of \( n \) \((m_1)\) from its old size, then adding the new size of \( n \) \((m_1 + pl)\). It is obvious that \( m_1 \) disappears from this equation, so that the new size of this is \( pt + pl \).

The return statement is handled in \( E6 \) and is simply:

\[
\left\{ \begin{array}{l}
1 : (\text{List}, pl) \\
n : (\text{List}, m_1 + pl) \\
n \prec\) this.next \\
\text{this} : (\text{List}, pt + pl)
\end{array} \right\} \quad \text{Return II}
\]

\[
\left\{ \begin{array}{l}
1 : (\text{List}, pl) \\
n : (\text{List}, m_1 + pl) \\
n \prec\) this.next \\
\text{this} : (\text{List}, pt + pl)
\end{array} \right\}
\]

When we derive the static sized type of this from its dynamic sized type using
4.1. LIST CONCATENATION

Axiom [1] we can see that for the then-branch of the conditional, the method annotation is correct:

\[ \text{this} : (\text{List}, pt + pl) \rightarrow \text{this} : [\text{List}, pt + pl] \]

We now have to continue with the else-branch, where \( n \) is null, which is handled in \( E4 \). Here, \( m_1 = 0 \) is added to \( \Omega \).

\[
\begin{align*}
\text{List} \prec \text{List} & \quad A(\text{List}, \text{next}) = \text{List} & \quad B(\text{List}, \text{next}) = 1 \\
\{ & \quad 1 : (\text{List}, pl) \\
& \quad \text{this} : (\text{List}, pt) \\
& \quad \text{this}.next : (\text{List}, m_1) \\
& \quad n : (\text{List}, m_1) \\
& \quad n \prec \text{this}.next \\
& \quad l \not\prec \text{this} \} & \quad \text{this}.next \leftarrow l & \quad l \not\prec \text{this} \\
& \quad \Omega_2(m_1=0) \end{align*}
\]

Let

\[
\begin{align*}
\text{let } \text{List} & = \text{this}.next \leftarrow l \text{ in } \text{return } (Q_2)_{\Omega_2} \\
\end{align*}
\]

Where \( E7 \) is the field update and \( E8 \) is the return statement. By assumption, this is not reachable from \( l \), because the parameters of a method must be totally separated. This information is needed because \( l \not\prec \text{this} \) is in the precondition of the field update rule to prevent the creation of circular data.

To handle the field update we have to use the frame rule again, in this case \( \text{UpdFrame} \). The set \( \text{modifies}(\text{this}.next \leftarrow l) \) is \( \{\text{this}.next\} \). We derive the sized type of \( \text{this}.next \) using Lemma [1]. The sized type of \( n \) is kept as well, because the old sized type of \( \text{this}.next \) is lost in the field update rule. Everything has to be in \( P \) here, because the entire condition is needed for the Update II rule.

\[
\begin{align*}
\text{List} \prec \text{List} & \quad A(\text{List}, \text{next}) = \text{List} & \quad B(\text{List}, \text{next}) = 1 \\
\{ & \quad 1 : (\text{List}, pl) \\
& \quad \text{this} : (\text{List}, pt) \\
& \quad \text{this}.next : (\text{List}, m_1) \\
& \quad n : (\text{List}, m_1) \\
& \quad n \prec \text{this}.next \\
& \quad l \not\prec \text{this} \} & \quad \text{this}.next \leftarrow l & \quad l \not\prec \text{this} \\
& \quad 1 : (\text{List}, pl) \\
& \quad \text{this} : (\text{List}, pt - m_1 + pl) \\
& \quad n : (\text{List}, m_1) \\
& \quad n \prec \text{this}.next \\
& \quad \text{result} \prec \text{this} \}
\end{align*}
\]

The proof for the return statement \( (E8) \) is just as basic as \( E6 \) (the \( \text{result} \) of the field update is discarded).

\[
\begin{align*}
\text{Return II} & \quad \text{let } \text{List} & = \text{this}.next \leftarrow l \text{ in } \text{return } (Q_2)_{\Omega_2} \\
\end{align*}
\]
In $\Omega$ the known value of $m_1$, 0, is stored. So we see that also in the else-branch
the size of this has become equal to $pt + pl$, so that the annotation is also correct
for this branch and thus for the entire method.

4.2 Tree to List Transformation

As a second example we will consider a method which transforms a tree structure
into a list. The list that the method returns will have the same amount of elements
as the tree.

We use the definition of the classes Tree and Node as given in Listing 3.1, which
is repeated here for readability in Listing 4.2. We use the same definition for lists
as in the previous section, i.e. that in Listing 4.1.

```java
class Tree {
    Object elem;
}

class Node extends Tree {
   /*@ count @*/ Tree left;
   /*@ count @*/ Tree right;
}
```

Listing 4.2: Tree structure in FJEU

The definitions of the methods leafToList() (in the class Tree) and nodeToList()
(in the class Node) are given in Listing 4.3 and Listing 4.4, respectively. The class
Tree stores a leaf. In regular Java a common implementation of a tree is to have
one abstract class Tree and extend this with classes Leaf and Node. In FJEU
there is no such thing as abstract classes, so it is left to the programmer not to
instantiate such a class. Therefore we chose to use the Tree as leaves as well,
instead of defining a separate class for this.

The correctness of the annotations for the leafToList() and nodeToList() methods
will be proved in Section 4.2.1 and Section 4.2.2 respectively. In Section 4.2.3
a different use of the logic will be shown. It will there be applied to the main()
method of a program, not to show that an annotation is correct, but to calculate
the actual sizes of constructed data structures.

4.2.1 Annotation Correctness for the leafToList() Method

In this section we will check the correctness of the annotations of the leafToList() method in the Tree class, as defined in Listing 4.3. An instantiated version of
the class Tree (not its subclass Node) is always a leaf, therefore the method is named leafToList(). Because a Tree object that is not a Node has no fields that are counted its size is always 1.

```plaintext
/*@ ensures ([sizetype] result; List{1}); */
List leafToList() {
  let List l = new List in
  let Object e = this.eim in
  return l.eim <- e;
}
```

Listing 4.3: The leafToList() method in FJEU

**Verifying the Aliasing Annotations** There are no method invocations in this method, so we can skip this step.

**Ensuring No Misinformation is Used** Although l is constructed at the first line of the method, it may not be considered local, because it is returned on the last line. The variable e is not local either. However, because elem is not a counted field of Tree, the change in aliasing by the field update on line 5 is still harmless to the logic.

**Verifying the Size Annotations** The body of the leafToList() method is simple, it consists of only three lines of code that construct a new List object, copy the element to it and return it. Its annotation has no precondition. It just specifies that the output of the method is always a list of length 1. We will check the correctness of this annotation against the strict-size logic.

We first consider the let expression on the first line of the method. The postcondition \( Q_1 \) and the class \( c_1 \) of the result of the new constructor are not yet known at this point.

\[
c_1 <\text{List } E1 \quad E2 \\
\{\text{this : [Tree, pt]}\} \text{let List l = new List in } \ldots \{Q_1\} \text{ Let}
\]

Where \( E1 \) is the new rule:

\[
\{\text{this : [Tree, pt]}\} \text{ new List } \begin{cases} \\
  \text{this : [Tree, pt]} \\
  \text{result : (List, 1)} \\
\end{cases} \text{ New}
\]
It can now be observed that result is an object of class List and that therefore c₁ = List and the class ontology relation c₁ <: List holds.

The result is bound to l in the precondition of E₂. The postcondition Q₁ is the same as that of the first let-rule and, indeed, as the postcondition of the entire method. The class c₂ is not yet known at this point, but note that the relation c₂ <: Object will hold for any c₂, because every class is a subclass of Object by definition.

\[ c₂ <: \text{Object} \quad E₃ \quad E₄ \]

Let

\[
\begin{align*}
\{ \text{this} : [\text{Tree, pt}] \} & \quad \text{let Object e = this.element in } \ldots \{ Q₁ \}
\end{align*}
\]

Where E₃ is the field access rule:

\[
\begin{align*}
\text{elem} & \in \text{Fields(Tree)} \\
\{ \text{this} : [\text{Tree, pt}] \} \quad \text{this.element} \quad \{ \text{this} : [\text{Tree, pt}] \} \quad \text{l : (List, 1)} \quad \{ \text{result} \leftarrow \text{this.element} \}
\end{align*}
\]

The return statement is handled in E₄, where result has been bound to e.

\[
\begin{align*}
\{ \text{this} : [\text{Tree, pt}] \} \quad \{ \text{l : (List, 1)} \} \quad \{ \text{e \leftarrow this.element} \} \quad \text{return l.element} \leftarrow e \{ Q₁ \}
\end{align*}
\]

Again, the postcondition Q₁ is equal to that of the let-rules and the entire method. It is also equal to the postcondition of E₅, so the correctness of the annotation depends only on the field update in E₅ now. This is an example where the Update I rule is used, because the field elem is not counted in the calculation of the size of a list. The Update I rule can be used without the frame rule, because it does not adapt sizes.

For the Update I rule we need the dynamic sized type of e. We can derive its static sized type by using Axiom 2 and Lemma 2:

\[
\begin{align*}
\text{this} : [\text{Tree, pt}] \rightarrow \text{this.element} : [\text{Object, m₁}] \\
\text{this.next} \leftarrow \text{e} \land \text{this.next} : [\text{Object, m₂}] \rightarrow \text{e} : [\text{Object, m₃}]
\end{align*}
\]

In the application of Axiom 2, m₁ = old(nthis.element) is added to Ω.

We would now need to apply Axiom 1 and finish the rest of our proof for every
subclass of \( \text{Object} \). But because in the Update I rule this dynamic type is only used to ensure that the class of \( e \) is a subclass of \( \text{Object} \), which is always true, we will here only show the case where the dynamic sized type of \( e \) is \( e: (\text{Object}, m_1) \).

We know that \( e \not\rightarrow l \) because \( l \) has no aliases and there is (obviously) no access path from \( e \) to \( l \).

\[
\begin{align*}
\text{Object} & \ll: \text{Object} \\
A(\text{List}, \text{elem}) & = \text{Object} \\
B(\text{List}, \text{elem}) & = 0 \\
\text{Update I} & \\
\begin{cases}
\quad l: (\text{List}, 1) \\
\quad e: (\text{Object}, m_1) \\
\quad \text{this} : [\text{Tree}, pt] \\
\quad e \leftarrow \text{this}.\text{elem} \\
\quad e \not\rightarrow l
\end{cases} & \quad \begin{cases}
\quad l: (\text{List}, 1) \\
\quad e: (\text{Object}, m_1) \\
\quad \text{this} : [\text{Tree}, pt] \\
\quad e \leftarrow \text{this}.\text{elem} \\
\quad l.\text{elem} \leftarrow e \\
\quad \text{result} \leftarrow l
\end{cases}
\end{align*}
\]

We can now derive the static sized type of \( \text{result} \) using Lemma \([1]\) and Axiom \([1]\)

\[
\begin{align*}
l & \leadsto \text{result} \land l : (\text{List}, 1) \rightarrow \text{result} : (\text{List}, 1) \\
\text{result} : (\text{List}, 1) & \rightarrow \text{result} : [\text{List}, 1]
\end{align*}
\]

So we see that the result of the method is indeed a list of size 1, i.e. that the annotation is correct.

### 4.2.2 Annotation Correctness for the nodeToList() Method

Next we will prove the correctness of the annotation of the \texttt{nodeToList()} method in the \texttt{Node} class. Because this is a long method with a lot of repetition we will skip some parts of the proof. We trust that the reader will be able to construct these parts herself.

**Verifying the Aliasing Annotations** Three different methods are invoked: \texttt{append()}, \texttt{leafToList()} and \texttt{nodeToList()} itself. We have validated the aliasing annotation of the \texttt{append()} method in Section \([4.1]\). The other two invoked methods both have no aliasing annotations, so we have to verify that they indeed do not change any aliasing relations between their parameters.

We will start with \texttt{leafToList()}. There is one field update in this method (and no method invocations). This field update involves two non-local variables, but because one of them (\(e\)) is not counted, a \texttt{mayaffectaliasing} annotation is not necessary.

Verifying that \texttt{nodeToList()} does not need an aliasing annotation takes a larger
CHAPTER 4. EXAMPLES OF ANNOTATION VERIFICATION

There is one field update in the method, which is identical to the field update in the `leafToList()` method and for the same reason does not necessitate an aliasing annotation. There are no less than eight method invocations in the `nodeToList()` method, but luckily the same reasoning is valid for all four combinations of method calls (i.e. the then- and else-branches of both conditionals).

Because the `leafToList()` and `nodeToList()` methods both have no aliasing annotations (and the latter is assumed to be correct) between `result` and a parameter (in fact they have no aliasing annotations at all), the variables `ll` and `lr` may be considered local to their respective branches. The invocations of `append()` therefore cannot affect the aliasing between multiple variables that are reachable from parameters. So, we can conclude that the `nodeToList()` method indeed does not need an `mayaffectaliasing` annotation.

```java
/*@ requires (\sizetype this; Node{pt}); */
/*@ ensures (\sizetype this; Node{pt}) */
/*@ && (\sizetype \result; List{pt}); */
/*@*/
List nodeToList() {
    let List lst = new List in
    let Object e = this.e in
    let List _ = l.elem <- e in
    let Tree l = this.left in
    let Tree r = this.right in

    if l instanceof Node then
        let List ll = l.nodeToList() in
        let List _ = lst.append(ll) in null
    else if l instanceof Tree then
        let List ll = l.leafToList() in
        let List _ = lst.append(ll) in null
    else
        let List _ = l in null

    if r instanceof Node then
        let List lr = r.nodeToList() in
        let List _ = lst.append(lr) in null
    else if l instanceof Tree then
        let List lr = r.leafToList() in
        let List _ = lst.append(lr) in null
    else
        let List _ = l in null

    return lst;
}
```

Listing 4.4: The `nodeToList()` method in FJEU
4.2. TREE TO LIST TRANSFORMATION

Ensuring No Misinformation is Used  Following the same reasoning as in the previous paragraph we can easily see that the only location where two non-local variables are affected by a field update or method invocation is on line 8, but since e is not counted this is harmless.

Verifying the Size Annotations  According to the annotation, the nodeToList method outputs a list of the exact same size as the tree the method is invoked on. We will now check this against the strict-size logic.

Because Node has no subclasses, the dynamic sized type of this can easily be derived from its static sized type using Axiom 1:

\[
\text{this} : [\text{Node, pt}] \rightarrow \text{this} : (\text{Node, pt})
\]

Lines 6 to 8 are almost identical to the leafToList() method, so we will skip ahead to the let expression on line 10 and assume that the reader can understand how the precondition for the let-rule was derived. It must also be noted that \( \Omega \) contains \( m_1 = \text{old}(\text{this}.\text{elem}) \) at this point.

Let

\[
\begin{align*}
\text{c}_1 & <: \text{Tree} \\
\{ & \text{lst} : (\text{List, 1}) \\
 & \text{e} : (\text{Object, } m_1) \\
 & \text{this} : (\text{Node, pt}) \\
 & \text{lst}.\text{elem} \leftarrow \text{e} \\
 & \text{e} \leftarrow \text{this}.\text{elem} \} \\
\end{align*}
\]

Let

\[
\begin{align*}
\text{next} & \in \text{Fields(Node)} \\
\{ & \text{lst} : (\text{List, 1}) \\
 & \text{e} : (\text{Object, } m_1) \\
 & \text{this} : (\text{Node, pt}) \\
 & \text{lst}.\text{elem} \leftarrow \text{e} \\
 & \text{e} \leftarrow \text{this}.\text{elem} \} \\
\end{align*}
\]

The class type of result in the postcondition of \( E_1 \) is not yet known, so it is represented by variable \( \text{c}_1 \) here. The postcondition \( Q_1 \) is the same one as that of the first three logical rules and of the entire method. Again, \( E_1 \) is considered first:

\[
\begin{align*}
\text{Access} \\
\{ & \text{lst} : (\text{List, 1}) \\
 & \text{e} : (\text{Object, } m_1) \\
 & \text{this} : (\text{Node, pt}) \\
 & \text{this}.\text{left} \\
 & \text{lst}.\text{elem} \leftarrow \text{e} \\
 & \text{e} \leftarrow \text{this}.\text{elem} \\
 & \text{result} \leftarrow \text{this}.\text{next} \}
\end{align*}
\]

The result of the field access is now bound to l by the let-rule. But we still have to check that \( \text{c}_1 \) is indeed a subclass of Tree. We can derive the static sized type of result by successively applying Axiom 2 and Lemma 2.
this : (Node, pt) → this.left : [Tree, m_2]
this.left ↝ result ∧ this.left : [Tree, m_2] → result : [Tree, m_2]

In the application of Axiom 2, m_2 = old(n_{this.left}) is added to Ω.

From the static sized type of result it is now clear that c_1 is either Tree or any of its subclasses and therefore that c_1 <: Tree holds.

It is easy to see that the next line (let Tree r = this.right in ...) has the same effect as the last one, and that the same rules should be used. Therefore they will be omitted for the sake of simplicity. The results of the two access rules are bound to l and r respectively in the next precondition. At this point Ω is {m_1 = old(n_{this.elem}), m_2 = old(n_{this.left}), m_3 = old(n_{this.right})}.

There is now a sequence of two conditionals and a return statement left. Therefore the sequence rule will be applied:

\[
\begin{array}{c}
\text{if ... if ... return lst; } \{Q_1\}
\end{array}
\]

The first conditional is handled in E3. Because this conditional branches on the class type of l we now have to split the static sized type of l up into all possible dynamic types, using Axiom 1

l : [Tree, m_2] → l : (Tree, m_2) ∨ l : (Node, m_2)

\[
\begin{array}{c}
\text{if } l \text{ instanceof Node then ... else ... } \{Q_2\}_\Omega
\end{array}
\]
4.2. TREE TO LIST TRANSFORMATION

The postcondition \( Q_2 \) must be derivable from the postconditions of \( E5, E6 \) and \( E7 \). We will return to this later, because \( Q_2 \) will become the precondition of \( E4 \).

We will handle \( E5 \) now first, in which \( m_2 \geq 1 \) is added to \( \Omega \):

\[
\begin{array}{l}
\text{c}_2 < : \text{List} & \text{E8} & \text{E9} \\
\begin{cases}
\text{lst} : \text{(List, } 1) \\
\text{e} : \text{(Object, } m_1) \\
\text{this} : \text{(Node, } pt) \\
\text{lst}.\text{elem} \leftarrow \text{e} \\
\text{e} \leftarrow \text{this}.\text{elem} \\
\text{l} \leftarrow \text{this}.\text{next} \\
\text{l} : \text{(Node, } m_2) \\
\text{r} : \text{[Tree, } m_3]
\end{cases}
\end{array}
\]

\[
\text{let List } ll = \text{l.nodeToList()} \text{ in } \ldots \{ Q_3 \}_{\Omega_{2}}.
\]

The recursive invocation of \( \text{l.nodeToList()} \) is handled in \( E8 \). As always, this should be done in conjunction with the frame rule.

The set \( \text{modifies(l.nodeToList())} \) is \( \emptyset \). Its calculation is left to the reader.

This means that, apart from the dynamic sized type of \( l \) which is needed by the method invocation rule, everything can be in \( R \).

\[
\Sigma(\text{Node, nodeToList()}) = \left\{ \begin{array}{l}
\text{this} : \text{(Node, } pt) \\
\text{result} : \text{List, } pt
\end{array} \right\}
\]

\[
\text{InvFrame}
\]

\[
\begin{array}{l}
\begin{cases}
\text{l} : \text{(Node, } m_2) \\
\text{lst} : \text{(List, } 1) \\
\text{e} : \text{(Object, } m_1) \\
\text{this} : \text{(Node, } pt) \\
\text{lst}.\text{elem} \leftarrow \text{e} \\
\text{e} \leftarrow \text{this}.\text{elem} \\
\text{l} \leftarrow \text{this}.\text{next} \\
\text{r} : \text{[Tree, } m_3]
\end{cases}
\end{array}
\]

\[
\begin{array}{l}
\begin{cases}
\text{l} : \text{(Node, } m_2) \\
\text{result} : \text{List, } m_2
\end{cases}
\end{array}
\]

We can now see that \( c_2 = \text{List} \), so that \( c_2 < : \text{List} \) holds. The \( \text{result} \) of the invocation is bound to \( ll \) by the let-rule in the precondition of \( E9 \), which is again a let-rule:
The invocation of `append()` is handled in $E_{10}$, again by a conjunction of the method invocation rule with the frame rule. We have shown in Section 4.1 that \(\text{modifies}(\text{lst.append}(ll)) = \{\text{lst}\}.$
### 4.2. TREE TO LIST TRANSFORMATION

The postcondition of the null-rule above is equal to $Q_3$.

We will now continue with the else-branch of the conditional, which is handled in $E6$ for the case where $l \neq \text{null}$. The case where $l = \text{null}$ is handled later in $E7$. In $E6$, $m_2 \geq 1$ is added to $\Omega$.

The else-branch of the first conditional starts out with a nested conditional.

<table>
<thead>
<tr>
<th>lst : (List, 1)</th>
<th>if</th>
</tr>
</thead>
<tbody>
<tr>
<td>l : (List, $m_2$)</td>
<td>if l instanceof Tree then ... else ... ($Q_4$) $\cup (m_2 \geq 1)$</td>
</tr>
<tr>
<td>lst.elem $\ni$ e</td>
<td></td>
</tr>
<tr>
<td>e $\ni$ this.elem</td>
<td></td>
</tr>
<tr>
<td>l : (Node, $m_2$)</td>
<td></td>
</tr>
<tr>
<td>e : (Object, $m_1$)</td>
<td></td>
</tr>
<tr>
<td>this : (Node, pt)</td>
<td></td>
</tr>
<tr>
<td>l $\ni$ this.next</td>
<td></td>
</tr>
<tr>
<td>r : [Tree, $m_3$]</td>
<td></td>
</tr>
</tbody>
</table>

Because at this point we know that $l$ is of dynamic type Tree and that its size $m_2 \geq 1$ (from $\Omega$), the preconditions of the latter two rules above the line of the if rule can never become true.

Because $E12$ is similar to $E5$ it is left to the reader. We will only show its postcondition, $Q_4$ here:
CHAPTER 4. EXAMPLES OF ANNOTATION VERIFICATION

\[ Q_4 = \{ \begin{align*}
\text{lst} : & \ (\text{List}, 1 + 1) \\
\text{ll} : & \ (\text{List}, 1) \\
\text{lst} . \text{elem} \not\sim e \\
e : & \ (\text{Tree}, m_2) \\
l : & \ (\text{Object}, m_1) \\
\text{this} : & \ (\text{Node}, pt) \\
l \not\sim \text{this.next} \\
r : & \ [\text{Tree}, m_3] \\
\text{result} : & \ (c_4, 0)
\end{align*} \]

We continue with \( E7 \) now, which is the case where \( l = \text{null} \). Therefore \( m_2 = 0 \) is added to \( \Omega \).

\( E7 \) is a let statement in which the value of \( l \) is output and then discarded, followed by a null. In other words, the code does nothing. It is the FJEU equivalent of not having an else-branch, which is not possible in the language. Its postcondition \( Q_5 \) is thus:

\[ Q_5 = \{ \begin{align*}
\text{lst} : & \ (\text{List}, 1) \\
\text{lst} . \text{elem} \not\sim e \\
e : & \ (\text{Tree}, m_2) \\
l : & \ (\text{Node}, m_2) \lor \ (\text{Tree}, m_2) \\
e : & \ (\text{Object}, m_1) \\
\text{this} : & \ (\text{Node}, pt) \\
l \not\sim \text{this.next} \\
r : & \ [\text{Tree}, m_3] \\
\text{result} : & \ (c_5, 0)
\end{align*} \]

We now have to find a common postcondition \( Q_2 \), which is derivable from \( Q_3, Q_4 \) and \( Q_5 \).

We know that in \( Q_4 \), \( l \) is a \text{Tree} and from \( \Omega \) we see that \( m_2 \geq 1 \). So, using Definition 1 we can derive that \( m_2 = \Delta(l, \text{Tree}) = 1 \). We can thus substitute \( \text{lst} : (\text{Tree}, 1 + 1) \) by \( \text{lst} : (\text{Tree}, 1 + m_2) \).

In \( Q_5 \), \( l = \text{null} \). Therefore, again, we can use Definition 1 to derive that \( m_2 = 0 \) and substitute \( \text{lst} : (\text{Tree}, 1) \) by \( \text{lst} : (\text{Tree}, 1 + m_2) \).

We can use Axiom 1 in the opposite direction to merge the two possible dynamic types back into one static type.
4.2. TREE TO LIST TRANSFORMATION

We now have our common postcondition $Q_2$, in which we see that $\text{lst}$ is now a list of length 1 plus the size of the left subtree of this. This is the postcondition of the first conditional.

\[
Q_2 = \begin{cases} 
\text{lst} : (\text{List}, 1 + m_2) \\
\text{lst}.\text{elem} \leftarrow e \\
e \leftarrow \text{this}.\text{elem} \\
l : [\text{Tree}, m_2] \\
e : (\text{Object}, m_1) \\
\text{this} : (\text{Node}, pt) \\
l \leftarrow \text{this}.\text{next} \\
r : [\text{Tree}, m_3] 
\end{cases}
\]

Because the proof for the second conditional is almost identical we leave that to the reader and skip ahead to the return statement at the end of the method. The single difference between the pre and postcondition of the first conditional is that the size of $\text{lst}$ is increased by $m_2$. In the second conditional, this is what happens with $m_3$.

\[
E_{13} \quad \text{return } \text{lst} ; \{Q_1\}
\]

Where $E_{13}$ is:
We see now that the size of \( \text{this} \) is indeed still \( pt \), so that part of the annotation is correct. We have to look into \( \Omega \) to be able to check the other part of the annotation. Because the precondition holds before execution of the method, it relates to the oldest stores of the \( n_{x,2} \) values that is present in \( \Omega \). In this case each is stored only once.

Before execution of the method, by Definition 4.2.3, \( pt = 1 + n_{\text{this}, \text{left}} + n_{\text{this}, \text{right}} \). We can now retrieve the oldest versions of \( n_{\text{this}, \text{left}} \) and \( n_{\text{this}, \text{right}} \) from \( \Omega \) and substitute, which gets us \( pt = 1 + m_2 + m_3 \). This is indeed the size of \( \text{result} \), so the annotation is correct.

### 4.2.3 Application to the Main Method

In this section, a different use of the strict-size logic is demonstrated. It is here applied to the main method to calculate the actual sizes of used data structures. This information could potentially be used to calculate the amount of heap space the program consumes. The main method is shown in Listing 4.5.

We start with an empty precondition.

\[
\text{Node} <: \text{Node} \quad \{ \} \quad \text{new Node} \{ \text{result} : (\text{Node}, 1) \} \quad \text{New} \quad E_1 \quad \text{Let}
\]

Where \( E_1 \) is:

\[
\text{Tree} <: \text{Tree} \quad \{ t : (\text{Node}, 1) \} \quad \text{new Tree} \quad \{ t : (\text{Node}, 1) \} \quad \text{New} \quad E_2 \quad \text{Let}
\]
4.2. TREE TO LIST TRANSFORMATION

```java
class Main {
    void main() {
        let Node t = new Node in
        let Tree l = new Tree in
        let Tree _ = t.left ← l in
        let Node r = new Node in
        let Tree _ = t.right ← r in
        let Tree rl = new Tree in
        let Tree _ = r.left ← rl in
        let Tree rr = new Tree in
        let Tree _ = r.right ← rr in
        let List lst = t.nodeToList() in
        return;
    }
}
```

Listing 4.5: Tree to list method invocation from main in FJEU

Where $E_2$ is:

$$
\text{Tree} <- \text{Tree} \quad A(\text{Node}, \text{left}) = \text{Tree} \quad B(\text{Node}, \text{left}) = 1
$$

In the UpdFrame rule above we know that the size of $t$.left is 0 before the update, because according to Definition $\Delta(t, \text{Tree}) = 1 + n_{t.left} + n_{t.right} = 1$. Because sizes are non-negative both subtrees must be null.

The derivation for lines 6 to 11 is similar, so we finally get:

$$
\text{List} <- \text{List} \quad E_4 \quad E_5
$$
CHAPTER 4. EXAMPLES OF ANNOTATION VERIFICATION

Where $E_4$ is:

$$
\Sigma(\text{Node, nodeToList}) = \left\{ \begin{array}{l}
\text{this} : (\text{Node, pt}), \\
\text{result} : (\text{List, pt})
\end{array} \right. 
\right\}
$$

And $E_5$ is:

$$
\begin{array}{l}
\text{t} : (\text{Node, 5}) \\
\text{l} : (\text{Tree, 1}) \\
\text{r} : (\text{Node, 3}) \\
\text{rl} : (\text{Tree, 1}) \\
\text{rr} : (\text{Tree, 1}) \\
\text{lst} : (\text{List, 5})
\end{array}
\rightarrow
\begin{array}{l}
\text{t} : (\text{Node, 5}) \\
\text{l} : (\text{Tree, 1}) \\
\text{r} : (\text{Node, 3}) \\
\text{rl} : (\text{Tree, 1}) \\
\text{rr} : (\text{Tree, 1}) \\
\text{lst} : (\text{List, 5})
\end{array}
$$

So we see that to execute this program we would need at least enough heap space to store a tree of size 5 and a list of size 5.

4.3 List Reversal

In this section we will verify the annotation for a method which reverses a list.

The reverse() method is shown in Listing 4.6

Verifying the Aliasing Annotations Two different methods are invoked from reverse(): append() and reverse() itself. We have verified the aliasing annotation of append() in Section 4.1.

To verify that reverse() does indeed not need an aliasing annotation, we look for field updates and method invocations where the aliasing between pairs of non-local variables may be affected. This is the case in the field update on line 9, but lst.elem is not counted. Even though reverse() does not have an aliasing annotation
4.3. LIST REVERSAL

involving \textit{result}, the variable \textit{nr} on line 13 may not be considered local, because it is returned at the end of the branch. However, in this branch the variable \textit{lst} is not returned (and it is constructed at the beginning of the method, so only a return would make it non-local), so it may be considered local in this branch. In the else-branch \textit{lst} is not local, but there aliasing is not changed between \textit{lst} and another non-local variable.

So, the method \texttt{reverse()} indeed does not change aliasing between its parameters and/or \textit{result}.

```
/*@ */
@ requires (\sizetype \this; List\{pt\})
@ ensures (\sizetype \this; List\{pt\})
@   && (\sizetype \result; List\{pt\})
/*@*/
List reverse() {
  let List lst = new List in
  let Object e = this.elem in
  let List _ = lst.elem <- e in

  let List n = this.next in
  if n instanceof List then
    let List nr = n.reverse() in
    let List _ = nr.append(lst) in
    return nr
  else
    return lst;
}
```

Listing 4.6: List reversal in FJEU

\textbf{Ensuring No Misinformation is Used} Following the same reasoning as in the previous paragraph we can easily see that the only location where two non-local variables are affected by a field update or method invocation is on line 9, but since \textit{e} is not counted this is harmless.

\textbf{Verifying the Size Annotations} The first three lines are exactly the same as those of the \texttt{nodeToList()} method, as described in Section 4.2.2 and very similar to those of the \texttt{leafToList()} method discussed in Section 4.2.1 so we will skip immediately to line 11.
The case where the class of there are three rules above the line in the definition of the if rule, but because holds, so we can continue with From the type of contained In the application of Axiom 2, result this this this Lemma 2 and Axiom 1:

\[
\begin{align*}
\text{let } \text{List } n & = \text{this}.\text{next} \ldots \{Q_1\} \\
\text{lst} &: (\text{List}, 1) \\
\text{e} &: (\text{Object}, m_1) \\
\text{this} &: (\text{List}, pt) \\
\text{lst}.\text{elem} &\leftarrow e \\
e &\leftarrow \text{this}.\text{elem} \\
\text{result} &\leftarrow \text{this}.\text{next}
\end{align*}
\]

To determine \(c_1\) and to be able to handle the conditional in \(E1\) we have to derive the dynamic sized type of \(n\). This is done by successively applying Axiom \(2\), Lemma \(2\) and Axiom \(1\):

\[
\begin{align*}
\text{this} &: (\text{List}, pt) \rightarrow \text{this}.\text{next} : [\text{List}, m_2] \\
\text{this}.\text{next} &\leftarrow \text{result} \land \text{this}.\text{next} : [\text{List}, m_2] \rightarrow \text{result} : [\text{List}, m_2] \\
\text{result} &: [\text{List}, m_2] \rightarrow \text{result} : (\text{List}, m_2)
\end{align*}
\]

In the application of Axiom \(2\) \(m_2 = old(n_{\text{this}.\text{next}})\) is added to \(\Omega\), which already contained \(m_1 = old(n_{\text{this}.\text{elem}})\).

From the type of \(\text{result}\) it is now clear that \(c_1\) is List and therefore that \(c_1 <\cdot:\)List holds, so we can continue with \(E1\).

\[
\begin{align*}
\text{lst} &: (\text{List}, 1) \\
\text{e} &: (\text{Object}, m_1) \\
\text{this} &: (\text{List}, pt) \\
n &: (\text{List}, m_2) \\
\text{lst}.\text{elem} &\leftarrow e \\
e &\leftarrow \text{this}.\text{elem} \\
n &\leftarrow \text{this}.\text{next}
\end{align*}
\]

If \(n\) instanceof List then \(\ldots\) else \(\ldots\) \(\{Q_1\}\), \(\Omega\)

There are three rules above the line in the definition of the if rule, but because the class of \(n\) is List, the precondition of the second rule can never become true.

The case where \(n\) is an instance of List is handled in \(E2\). We will continue with \(E3\)
(the case where \( n \) is null) after we have completed \( E2 \). In \( E2, \ m_2 \geq 1 \) is added to \( \Omega \).

\[
\begin{align*}
\Sigma(\text{List, reverse}) = & \quad \left\{ \text{this} : (\text{List}, \text{pt}) \right\} \quad \left\{ \text{result} : (\text{List}, \text{pt}) \right\} \\
& \quad \begin{aligned}
\text{InvFrame} & \end{aligned}
\end{align*}
\]

Because the size of \( n \) is not changed \( R' = R \). We see now that \( c_2 = \text{List} \), thus that \( c_2 < \text{List} \) holds.

Continuing with \( E5 \):
The return statement is handled in $E_7$.

Let $E_6$:

$$\begin{align*}
\{ n : (\text{List}, m_2), \\
nr : (\text{List}, m_2), \\
lst : (\text{List}, 1), \\
e : (\text{Object}, m_1), \\
this : (\text{List}, pt), \\
lst.\text{elem} \leftarrow e, \\
e \leftarrow \text{this}.\text{elem}, \\
n \leftarrow \text{this}.\text{next}\} \end{align*}$$

let $\text{List}_e \leftarrow \text{nr}.\text{append}(\text{lst})$ in $\ldots \{ Q_2 \}$

Where $E_6$ is again a conjunction of the method invocation rule with the frame
rule. As shown in Section 4.1 the set $\text{modifies}(\text{nr}.\text{append}(\text{lst})) = \{ \text{nr} \}$. Because
no other program variables are reachable from $\text{nr}$, the rest of the precondition can
be in $\bar{R}$.

$$\begin{align*}
\Sigma(\text{List} , \text{append}(l)) = \left\{ \begin{array}{l}
\{ l : (\text{List}, pl) \\
\text{this} : (\text{List}, pt) \} \\
\{ l : (\text{List}, pl) \\
\text{this} : (\text{List}, pt + pl) \} \end{array} \right\}
\end{align*}$$

InvFrame

$$\begin{align*}
\{ \begin{array}{l}
\text{lst} : (\text{List}, 1) \\
nr : (\text{List}, m_2) \\
n : (\text{List}, m_2) \\
e : (\text{Object}, m_1) \\
this : (\text{List}, pt) \\
lst.\text{elem} \leftarrow e, \\
e \leftarrow \text{this}.\text{elem}, \\
n \leftarrow \text{this}.\text{next} \end{array} \} \end{align*}$$

Because $\text{lst}$ and $\text{nr}$ are not reachable from any of the free variables in $R$, $R' = R$.

The return statement is handled in $E_7$:  

$$\begin{align*}
\{ n : (\text{List}, m_2), \\
nr : (\text{List}, m_2), \\
lst : (\text{List}, 1), \\
e : (\text{Object}, m_1), \\
this : (\text{List}, pt), \\
lst.\text{elem} \leftarrow e \\
en \leftarrow \text{this}.\text{next} \} \end{align*}$$
When we apply Lemma 4.3 we see that the result of this branch is a list of size $m_2 + 1$:

$$\text{nr} \sim \text{result} \land \text{nr} : (\text{List}, m_2 + 1) \rightarrow \text{result} : (\text{List}, m_2 + 1)$$

When we now apply Definition 4.3 to the annotation, we see that $pt = \Delta(\text{this}, \text{List}) = 1 + n_{\text{this.next}}$ (because $m_2 \geq 1 \in \Omega$), where the value of $n_{\text{this.next}}$ is that before execution of the method, i.e. the oldest value stored in $\Omega$, which is equal to $m_2$. If we substitute we see that $pt = 1 + m_2$, thus that the result of this method is a list of size $pt$ and the annotation is indeed correct for this branch.

We must continue with the else-branch now, which is handled in $E3$:  

$$\begin{align*}
\text{lst} &: (\text{List}, 1) \\
\text{nr} &: (\text{List}, m_2 + 1) \\
\text{n} &: (\text{List}, m_2) \\
\text{e} &: (\text{Object}, m_1) \\
\text{this} &: (\text{List}, \text{pt}) \\
\text{lst}.\text{elem} &: e \\
\text{e} &: \text{this}.\text{elem} \\
\text{n} &: \text{this}.\text{next}
\end{align*}$$

$$\begin{align*}
\text{lst} &: (\text{List}, 1) \\
\text{nr} &: (\text{List}, m_2 + 1) \\
\text{n} &: (\text{List}, m_2) \\
\text{e} &: (\text{Object}, m_1) \\
\text{this} &: (\text{List}, \text{pt}) \\
\text{lst}.\text{elem} &: e \\
\text{e} &: \text{this}.\text{elem} \\
\text{n} &: \text{this}.\text{next}
\end{align*}$$
no problem. This also means that on line 7, so, we need to take a look at the method invocations at lines 7, 10 and 11.

verify that polynomial( ) has been verified in Section 4.3 and Section 4.1 respectively. So we only have to reverse of its tail, the reverse of the tail of its tail, et cetera. If for example the a list as input and outputs the concatenation of the reverse of this list with the polynomial( ) takes a list as input and outputs the concatenation of the reverse of this list with the reverse of its tail, the reverse of the tail of its tail, et cetera. If for example the following list of integers was supplied as input: \([1, 4, 2, 6]\), then the output would be \([6, 2, 4, 1, 6, 2, 4, 6, 2, 6]\). The polynomial( ) method is shown in Listing 4.7.

Because \(\Omega \cup \{m_2 = 0\} \vdash pt = 1 + m_2 \land m_2 = 0\), we see that \(pt = 1\), thus that the annotation is correct for this branch as well.

4.4 Polynomial Example

In this section we will verify the annotation of a method which outputs a list of which the size is given by a non-linear polynomial. The method polynomial( ) takes a list as input and outputs the concatenation of the reverse of this list with the reverse of its tail, the reverse of the tail of its tail, et cetera. If for example the following list of integers was supplied as input: \([1, 4, 2, 6]\), then the output would be \([6, 2, 4, 1, 6, 2, 4, 6, 2, 6]\). The polynomial( ) method is shown in Listing 4.7.

Verifying the Aliasing Annotations Three methods are invoked: reverse(), polynomial( ) and append(). The aliasing annotations of reverse() and append() have been verified in Section 4.3 and Section 4.1 respectively. So we only have to verify that polynomial( ) itself indeed does not need an aliasing annotation. To do so, we need to take a look at the method invocations at lines 7, 10 and 11.

On line 7, reverse() is invoked. This method has no aliasing annotations, so poses no problem. This also means that lst does not have any aliasing relation with this.
4.4. POLYNOMIAL EXAMPLE

Listing 4.7: The polynomial() method in FJEU

However, it may not be considered local to this method, since it is returned at the end of both branches of the conditional.

On line 10, polynomial() is invoked. This method also has no aliasing annotations. This means that tl is local to the then-branch.

On line 11, append() is invoked. This method has a mayaffectaliasing annotation for the parameter pair lst and tl. Because tl is local to the then-branch, this does not necessitate an aliasing annotation for the polynomial() method.

Ensuring No Misinformation is Used Following the same reasoning as in the previous paragraph, it is clear that no incomplete information can exist in the logic.

Verifying the Size Annotations We start with the let expression on line 7:

\[
\begin{align*}
c_1 :& \text{: List} & E_1 & E_2 \\
\{ \text{this : (List, pt)} \} & \text{let List lst = this.reverse() in} & \ldots & \{Q_1\}
\end{align*}
\]

Where \( E_1 \) is the combination of a method invocation rule with the frame rule:
\[
\Sigma(\text{List, reverse()}) = \left\{ \begin{array}{l}
\text{this : (List, pt)} \\
\text{result : (List, pt)}
\end{array} \right\}
\]

\[
\begin{array}{l}
\{ \text{this : (List, pt)} \} \text{ this.reverse()}
\end{array}
\]

\[
\begin{array}{l}
\{ \text{this : (List, pt)} \} \text{ result : (List, pt)}
\end{array}
\]

So we see that \( c_1 = \text{List} \) and continue with \( E2 \):

\[
\begin{array}{l}
\text{next} \in \text{Fields(List)}
\end{array}
\]

\[
\begin{array}{l}
\{ \text{this : (List, pt)} \}
\{ \text{this.next : (List, pt)} \}
\end{array}
\]

\[
\begin{array}{l}
\{ \text{this : (List, pt)} \}
\{ \text{result : (List, pt)} \}
\end{array}
\]

\[
\begin{array}{l}
\text{let List n} = \text{this.next in} \ldots \{ Q_1 \}
\end{array}
\]

We successively apply Axiom 2, Lemma 2 and Axiom 1 to derive the dynamic type of \( \text{result} \), from which we can read that \( c_2 = \text{List} \), thus that \( c_2 : \text{List} \) holds.

\[
\begin{array}{l}
\text{this : (List, pt)} \rightarrow \text{this.next : [List, m_1]}
\end{array}
\]

\[
\begin{array}{l}
\text{this.next} \sim \text{result} \land \text{this.next : [List, m_1]} \rightarrow \text{result : [List, m_1]}
\end{array}
\]

\[
\begin{array}{l}
\text{result : [List, m_1]} \rightarrow \text{result : (List, m_1)}
\end{array}
\]

In the application of Axiom 2, \( m_1 = \text{old}(n_{\text{this.next}}) \) is added to \( \Omega \).

The conditional is handled in \( E3 \):

\[
\begin{array}{l}
\text{if n instanceof List then} \ldots \text{else} \ldots \{ Q_1 \}_\text{\Omega}
\end{array}
\]

There are three rules above the line in the definition of the if rule, but because the class of \( n \) is List, the precondition of the second rule can never become true.

The case where \( n \) is an instance of List is handled in \( E4 \). We will continue with \( E5 \) (the case where \( n \) is null) after we have completed \( E4 \). In \( E4 \), \( m_1 \geq 1 \) is added to \( \Omega \).
Where $E_6$ is again a conjunction of the method invocation rule with the frame rule. The set $\text{modifies}(n.\text{polynomial}())$ is $\emptyset$, we can therefore put everything apart from what is needed by the invoke rule in $R$.

Because the size of $n$ is not changed $R' = R$. We see now that $c_3 = \text{List}$, thus that $c_3 <:: \text{List}$ holds.

Continuing with $E_7$:

Where $E_8$ is again a conjunction of the method invocation rule with the frame rule. As shown in Section 4.1, the set $\text{modifies}(\text{lst.append}(tl))$ is $\{\text{lst}\}$. Because no other program variables are reachable from $\text{lst}$, the rest of the precondition can be in $R$. 

4.4. POLYNOMIAL EXAMPLE  

\[
\text{Σ} \{\text{List, polynomial}()\} = \left\{ \begin{array}{l}
\text{this} : (\text{List, pt}) \\
\text{result} : (\text{List, } 0.5 \cdot \text{pt} \cdot \text{pt} + 0.5 \cdot \text{pt})
\end{array} \right\}
\]

InvFrame

\[
\begin{array}{l}
\text{n} : (\text{List, } m_1) \\
\text{this} : (\text{List, pt}) \\
\text{lst} : (\text{List, pt}) \\
\text{n} \leftarrow \text{this.next}
\end{array}
\]

\[
\begin{array}{l}
\text{n} : (\text{List, } m_1) \\
\text{result} : (\text{List, } 0.5 \cdot m_1 \cdot m_1 + 0.5 \cdot m_1) \\
\text{this} : (\text{List, pt}) \\
\text{lst} : (\text{List, pt}) \\
\text{n} \leftarrow \text{this.next}
\end{array}
\]
Because \( \text{lst} \) and \( \text{tl} \) are not reachable from any of the free variables in \( R, R' = R \).

The return statement is handled in \( E9 \):

When we apply Lemma \( \PageIndex{1} \) we see that the result of this branch is a list of size
\( \text{pt} + 0.5 \cdot m_1 \cdot m_1 + 0.5 \cdot m_1 : \)

\[
\text{lst} \sim \text{result} \quad \land \quad \text{lst} : (\text{List, pt} + 0.5 \cdot m_1 \cdot m_1 + 0.5 \cdot m_1) \rightarrow
\]

\[
\text{result} : (\text{List, pt} + 0.5 \cdot m_1 \cdot m_1 + 0.5 \cdot m_1)
\]

When we now apply Definition \( \PageIndex{1} \) to the annotation, we see that \( \text{pt} = \Delta(\text{this, List}) = 1 + n_{\text{this.next}} \), where the value of \( n_{\text{this.next}} \) is that before execution of the method, i.e. the oldest value stored in \( \Omega \), which is equal to \( m_1 \), so \( \text{pt} = 1 + m_1 \). If we substitute and simplify:
4.4. POLYNOMIAL EXAMPLE

\[ pt + 0.5 \cdot m_1 \cdot m_1 + 0.5 \cdot m_1 = pt + 0.5 \cdot (pt - 1) \cdot (pt - 1) + 0.5 \cdot (pt - 1) \]
\[ = 0.5 \cdot (pt \cdot pt - 2 \cdot pt + 1) + 0.5 \cdot pt - 0.5 + pt \]
\[ = 0.5 \cdot pt \cdot pt - pt + 0.5 + 1.5 \cdot pt - 0.5 \]
\[ = 0.5 \cdot pt \cdot pt + 0.5 \cdot pt \]

So we see that the (polynomial) size annotation is indeed correct for this branch.

We must continue with the else-branch now, which is handled in E5. In E5, \( m_1 = 0 \) is added to \( \Omega \).

---

When we apply Lemma \[1\] we see that the result of this branch is a list of size \( pt \):

\[ \text{lst} \leftarrow \text{result} \land \text{lst} : (\text{List}, pt) \rightarrow \text{result} : (\text{List}, pt) \]

Because \( \Omega \cup \{ m_1 = 0 \} \models pt = 1 + m_1 \land m_1 = 0 \), we see that \( pt = 1 \). If we substitute we see that the annotation is also correct for this branch, because:

\[ pt = 0.5 \cdot pt \cdot pt + 0.5 \cdot pt = 0.5 \cdot 1 \cdot 1 + 0.5 \cdot 1 = 1. \]
Chapter 5

Soundness Proof Sketch

In this section a proof of the soundness of the strict-size logic with respect to the operational semantics, as specified in [25], will be sketched. To be able to relate the strict-size logic to the operational semantics, a series of definitions is supplied in Section 5.1. A soundness proof for the combination of the frame rule with the field update rule, which is the most important part of the strict-size logic, is then given in Section 5.2. Finally, Section 5.3 describes how this might be extended to a complete soundness proof.

5.1 Relating the Strict-Size Logic to the Operational Semantics

In order to relate the strict-size logic specified herein to the operational semantics of FJEU as given in [25], a series of definitions is supplied in this section.

Definition 4 specifies how the size of an object on the heap can be calculated. This is analogous to Definition 1 in the sense that the size of an object is either null or one plus the sum of the sizes of its counted attributes.

**Definition 4 (Calculating Size from the Heap):** The size of the object at location \( l \) on heap \( \sigma \) is equal to:

\[
\text{size}_\sigma(l) = \begin{cases} 
0 & \text{if } l = 0 \\
1 + \sum_{i=1}^{k} \text{size}_\sigma(v_i) \cdot B(C, a_i) & \text{if } \sigma_l = (C, a_1 : v_1, \ldots, a_k : v_k)
\end{cases}
\]

We use the definition of aliases as given in [25] (on page 5). For readability, it is repeated here.
Definition 5 (Aliasing (Operational Semantics)): “An alias \( \nu.\bar{p} \) is a pair of a stack value \( \nu \) and a (possibly empty) access path \( \bar{p} \). For a given heap \( \sigma \) we recursively define \( \llbracket \nu.\bar{p} \rrbracket \sigma \) by:

\[
\llbracket \nu.\bar{p} \rrbracket \sigma = \begin{cases} 
\nu & \text{if } \bar{p} = \emptyset \\
\nu_i & \text{if } \bar{p} = \bar{q}.a_i \land \sigma \llbracket \nu.\bar{q} \rrbracket = (C, a_1 : v_1, \ldots, a_k : v_k) \\
0 & \text{otherwise}
\end{cases}
\]

We say that \( \nu.\bar{p} \) is an alias for the location \( l \) within \( \sigma \) if \( \llbracket \nu.\bar{p} \rrbracket \sigma = l \) holds, i.e. the alias represents a valid path leading to a proper location.”

Now, using the definition for aliasing, the reachability relation can be defined recursively for the operational semantics.

Definition 6 (Reachability (Operational Semantics)): \( l_1 \sim_{\sigma} l_2 \) iff either

1. \( l_1 = l_2 \)

2. \( \sigma_{l_1} = (C, a_1 : v_1, \ldots, a_k : v_k) \land \bigwedge_{i=1}^k v_k \sim_{\sigma} l_2 \)

We assume that there is a special stack value \textit{result} that points to the result of an expression. We use the notation \( e \to l \) (\( e \) evaluates to \( l \)) instead of that used in [25] to avoid confusion with the notation for reachability.

Definition 7 (Result Pointer): Given that some expression \( e \) evaluates to \( l \): \( \eta, \sigma \vdash e \to l, \tau \), this means that \( \eta_{\text{result}} = l \).

For a given stack and heap configuration, we can verify if a condition from the strict-size logic holds (we say that it is \textit{valid} w.r.t. stack \( \eta \) and heap \( \sigma \)), by using the following definition. We use induction on the size of condition \( P \).

Definition 8 (Validity): Given a stack \( \eta \), heap \( \sigma \) and condition \( P \), \( \text{Valid}(\eta, \sigma, P) \) iff either

1. \( P = \emptyset \)

2. \( P = \{ x.\bar{a} : (C, p) \} \land \eta_x = v \land \llbracket v.\bar{a} \rrbracket \sigma = l \land \sigma_l = (C, a_1 : v_1, \ldots, a_k : v_k) \land \text{size}_\sigma(l) = p \)

3. \( P = \{ x.\bar{a} \to y.\bar{b} \} \land \eta_x = v \land \eta_y = w \land \llbracket v.\bar{a} \rrbracket \sigma = \llbracket w.\bar{b} \rrbracket \sigma \)

4. \( P = Q \cup R \land \text{Valid}(\eta, \sigma, Q) \land \text{Valid}(\eta, \sigma, R) \)

where \( \bar{a} \) and \( \bar{b} \) are possibly empty access paths.

We can now define soundness of the strict-size logic w.r.t. the operational semantics as follows.
Theorem 1 (Soundness): Given a stack $\eta$, heaps $\sigma$ and $\tau$, an expression $e$ and a heap location $l$, let $\eta, \sigma \vdash e \rightarrow l, \tau$. Then if $\eta, \sigma$ satisfies some precondition $P$ and some postcondition $Q$ can be derived from the strict-size logic, then $\eta, \tau$ satisfies this postcondition. Formally:

$$\text{Valid}(\eta, \sigma, P) \rightarrow \text{Valid}(\eta, \tau, Q)$$

To accommodate for simpler proofs, four additional lemmas are introduced. These all apply to cases where a separate part of the heap is constant and can be used to prove that for the objects in that part aliasing, reachability and sizes are also constant under certain conditions. An example heap configuration is shown in Figure 5.1.

![Figure 5.1: Example heap configuration](image)

In this example, if the objects at locations $v$ and $[v.a_i]_\sigma$ are the same for heaps $\sigma$ and $\tau$ (an object is a combination of a class $C$ with a set of heap location pointing to the attributes, for example $(C, a_i : v_1, \ldots, a_k : v_k)$), then $[v.a_i.a_k]_\tau$ must point to the same heap location as $[v.a_i.a_k]_\sigma$. These situations are captured by Lemma 4.

Lemma 4 (Constant Aliasing): $$(\forall \vec{b} \prec \vec{a} . \sigma_{[v.\vec{b}]_\sigma} = \tau_{[v.\vec{b}]_\tau}) \rightarrow [v.\vec{a}]_\sigma = [v.\vec{a}]_\tau$$

Proof By induction on the length $n$ of $\vec{a}$.

In the case that $n = 0$, then $\vec{a} = \epsilon$. Therefore also $\vec{b} = \epsilon$, so $v.\vec{a} = v.\vec{b} = v$. According to Definition 5, $[v]_\sigma = v$ for any heap $\sigma$. So, $[v]_\sigma = [v]_\tau$ is always true.

In the case that $n = m + 1$ and the lemma holds for an access path $v.\vec{c}$ with $|\vec{c}| = m$, then $\vec{a} = \vec{c}.a_i$. According to Definition 5, $[v.\vec{c}.a_i]_\sigma = v_i$ for a heap $\sigma$ if $\sigma_{[v.\vec{c}]} = (C, a_i : v_1, \ldots, a_k : v_k)$. This means that if $\sigma_{[v.\vec{c}]} = \tau_{[v.\vec{c}]}$, then $[v.\vec{c}.a_i]_\sigma = [v.\vec{c}.a_i]_\tau$.

We assume $\forall \vec{b} \prec \vec{c}.a . \sigma_{[v.\vec{b}]_\sigma} = \tau_{[v.\vec{b}]_\tau}$, so $\sigma_{[v.\vec{b}]_\tau} = \tau_{[v.\vec{b}]_\tau}$. What is left to prove now is that $\tau_{[v.\vec{c}]} = \tau_{[v.\vec{c}]}$.

In the induction we assume $(\forall \vec{d} \prec \vec{c}.a . \sigma_{[v.\vec{d}]_\sigma} = \tau_{[v.\vec{d}]_\tau}) \rightarrow [v.\vec{c}]_\sigma = [v.\vec{c}]_\tau$. Since we also know $\forall \vec{d} \prec \vec{c}.a . \sigma_{[v.\vec{d}]_\sigma} = \tau_{[v.\vec{d}]_\tau}$, then definitely $\forall \vec{d} \prec \vec{c}.a . \sigma_{[v.\vec{d}]_\sigma} = \tau_{[v.\vec{d}]_\tau}$, thus $[v.\vec{c}]_\sigma = [v.\vec{c}]_\tau$. \qed
When we look at Figure 5.1 we see that location \([v.a_1.a_k]_\sigma\) is reachable from location \(v\) in heap \(\sigma\). If the objects at all locations that can be reached from \(v\) are equal in heaps \(\sigma\) and \(\tau\), then \([v.a_1.a_k]_\tau\) is also reachable from \(v\) in heap \(\tau\). Such cases are captured by Lemma 5.

**Lemma 5 (Constant Reachability):** \(l_1 \sim_\sigma l_2 \land (\forall l' \in \text{loc} . l_1 \sim_\sigma l' \rightarrow \sigma_l = \tau_l) \rightarrow l_1 \sim_\tau l_2\)

**Proof** By induction on the length of the path from \(l_1\) to \(l_2\).

If the length of the path is zero, i.e. \(l_1 = l_2\), then we can read directly from Definition 6 that \(l_1 \sim_\tau l_2\).

If the length of the path is \(n\) and the lemma holds for a path of length \(n - 1\), then we know that there is a location \(l_n\), for which \(l_n \sim_\tau l_2\) and a location \(l_{n-1}\) with \(\sigma_{l_{n-1}} = (C, a_1 : v_1, \ldots, a_k : v_k)\), with \(v_i = l_n\) for some \(i\). Because \(\forall l' \in \text{loc} . l_1 \sim_\sigma l' \rightarrow \sigma_l = \tau_l, \sigma_{l_{n-1}} = \tau_{l_{n-1}}\). Thus by part 2 of Definition 3 we see that \(l_1 \sim_\tau l_2\).

If Figure 5.1 is complete, then \(\text{size}_\sigma(v) = 5\). If the objects at all locations that can be reached from \(v\) are equal in heaps \(\sigma\) and \(\tau\), then the size of \(v\) must also be 5 in heap \(\tau\). This is expressed by Lemma 6.

**Lemma 6 (Constant Size):** \((\forall l' \in \text{loc} . l \sim_\sigma l' \rightarrow \sigma_l = \tau_l) \rightarrow \text{size}_\sigma(l) = \text{size}_\tau(l)\)

**Proof** Because we assume that there is no circular data, a datastructure can always be seen as a directed acyclic graph. We will call nodes of the graph with null-pointers for all the attributes leaves. We will prove the lemma by induction on the maximal distance \(n\) to these leaves.

If this maximal distance is 0, i.e. \(\sigma_l = (C, a_1 : 0, \ldots, a_k : 0)\), then \(\text{size}_\sigma(l) = \text{size}_\tau(l) = 1\).

If this maximal distance \(n = m + 1\) and we know that \(\text{size}_\sigma(l_m) = \text{size}_\tau(l_m)\) for any location \(l_m\) with a maximal distance \(m\) to its leaves, then \(\text{size}_\tau(l) = 1 + \sum_{i=1}^k \text{size}_\tau(v_i) \cdot B(C, a_i)\), with \(\tau_l = (C, a_1 : v_1, \ldots, a_k : v_k)\). Because we know from \(\forall l' \in \text{loc} . l \sim_\sigma l' \rightarrow \sigma_l = \tau_l\) that \(\sigma_l = \tau_l\) and we also know that the maximal distance from any \(v_i\) to its leaves is \(m\) and thus \(\text{size}_\tau(v_i) = \text{size}_\sigma(v_i)\), this is equal to \(1 + \sum_{i=1}^k \text{size}_\sigma(v_i) \cdot B(C, a_i)\), with \(\sigma_l = (C, a_1 : v_1, \ldots, a_k : v_k)\), thus \(\text{size}_\sigma(l) = \text{size}_\tau(l)\).
5.2 Soundness of the Update/Frame Rule

As an example, the soundness proof for the Update II rule in conjunction with the frame rule is given in this section. This combination forms the kernel of the strict-size logic and is its most complex fraction.

As described in Section 3.2.10, the conjunction of the frame rule with the update or method invocation rules should be seen as a single rule. We will therefore prove soundness as such.

The conjunction of the Update II rule with the frame rule is specified as follows:

\[
\begin{align*}
E &: F \\
A(C, a) &= F \\
B(C, a) &= 1
\end{align*}
\]

\[\text{UpdFrame}\]

\[
\begin{aligned}
&x : (C, p_x) \\
x.a : (D, p_a) \\
y : (E, p_y) \\
x.a \mapsto z_1 \\
\vdots \\
x.a \mapsto z_k \\
z_i : (D, p_z) \\
y \not\in x \\
R
\end{aligned}
\]

\[
\begin{aligned}
x : (C, p_x - p_a + p_y) \\
y : (E, p_y) \\
x.a \mapsto y \\
z_i : (D, p_z) \\
\text{result} \not\in x \\
R'
\end{aligned}
\]

Where \(R'\) is constructed from \(R\) as described in 3.2.10. As does the frame rule, this combination of rules has the side condition:

\[
\forall r \in \text{FV}(R) \cdot \forall q \in \text{modif ies}(x.a \leftarrow y) \cdot q \not\in R
\]

The set \(\text{modif ies}(x.a \leftarrow y)\) consists of \(x.a\) and all its aliases \((z_1 \ldots z_k)\). This means that everything in the precondition that may not be a part of \(R\) is explicitly mentioned in this case.

The operational semantics rule for Update is specified in 25 as follows:

\[
\begin{align*}
\eta_k = l \\
\sigma_l = (C, a_1 : v_1, \ldots, a_k : v_k) \\
a = a_i \\
\tau = \sigma[l.a_i \mapsto \eta_i] \\
\eta, \sigma \vdash x.a \leftarrow y \rightarrow l, \tau
\end{align*}
\]

\[\text{OSUpdate}\]

Soundness of the UpdFrame rule now informally means that if the precondition (and in this case also the side condition) hold(s) for certain \(\eta, \sigma\), then the post-condition holds for \(\eta, \tau\).
We therefore assume a stack \( \eta \) and heap \( \sigma \) that are valid with respect to the precondition. We will write \( P_{uf} \) and \( Q_{uf} \) for the pre and postconditions of the UpdFrame rule, respectively.

We assume \( \text{Valid}(\eta, \sigma, P_{uf}) \), i.e. that (from Definition 8):

\[
\begin{align*}
\eta_x &= l_x \\
\{ l_x.a \}_\sigma &= l_a \\
\eta_y &= l_y \\
\eta_z &= w_1 \land \ldots \land \eta_{z_k} = w_k \land \{ w_1.a_{l_1} \}_\sigma = l_2 \land \ldots \land \{ w_k.a_{l_k} \}_\sigma = l_k \\
l_y & \not\sim_{\sigma} l_x
\end{align*}
\]

The sized types and aliasing relations over access paths \( r.a \) for which the predicate \( \{ r.a \}_\sigma = l \land l \not\sim_{\sigma} l_y \land l_y \not\sim_{\sigma} l \) holds are in \( R \), so we know that \( \forall r.a \in R : l \not\sim_{\sigma} [r.a]_\sigma = l_y \). From the operational semantics update rule we read that \( \tau \) is constructed from \( \sigma \) by making \( l_y.a \) a point at \( l_y \). So we know \( \tau_y = (C, a_1 : v_1, \ldots, a_y : l_y, \ldots, a_k : v_k) \). All other heap locations are unchanged: \( \forall l \in \text{loc} \cdot l \not\sim_{\sigma} l_y \Rightarrow \tau_y = \tau_l \).

From this collection of information we now need to prove \( \text{Valid}(\eta, \tau, Q_{uf}) \). Given part 4 of Definition 5 we can achieve this by proving the validity of each individual sized type and aliasing relation in \( Q_{uf} \). We start by proving validity of the sized types in \( Q_{uf} \), followed by the proof for the aliasing relations and an inductive proof that all sized types and aliasing relations in \( R' \) are valid.

**Sized Types** Because of the absence of circular data\(^1\) we know that \( l_y \not\sim_{\sigma} l_x \), thus \( l_y \not\equiv l_x \). Since \( \eta, \sigma \) satisfies the precondition we also know that \( l_y \not\sim_{\sigma} l_x \), thus that \( l_y \not\equiv l_x \).

From \( \forall l \in \text{loc} : l \not\sim_{\sigma} l_y \Rightarrow \tau_l = \tau_y \) we can derive \( \sigma_l = \tau_l \) and \( \sigma_y = \tau_y \).

Using Lemma 6 we can derive that \( \text{size}_\sigma(l_x) = \text{size}_\tau(l_x) \) and \( \text{size}_\sigma(l_y) = \text{size}_\tau(l_y) \).

When we combine this with Definition 8 we see that \( \text{Valid}(\eta, \tau, \{ y : (E, p_y) \}) \) and \( \forall i \in \{1, \ldots, k\} \cdot \text{Valid}(\eta, \tau, \{ z_i : (D, p_x) \}) \).

We can calculate \( \text{size}_\sigma(l_x) \) in terms of the sizes of other heap locations by applying Definition 4:

\[
\text{size}_\sigma(l_x) = 1 + \sum_{i=1}^{n-1} \text{size}_\sigma(v_i) \cdot B(C, a_i) + \text{size}_\sigma(l_x) + \sum_{j=n+1}^{k_x} \text{size}_\sigma(v_j) \cdot B(C, a_j)
\]

---

\( ^1 \)The proof for this is left to the reader, but it is obvious that circular paths can only be introduced by a field update on an object which points that field to a heap location that can be reached from that object, which is prohibited by the requirement of \( y \not\sim x \) in the precondition.
5.2. SOUNDNESS OF THE UPDATE/FRAME RULE

Where a is the n-th attribute of the object at location l. The multiplication with B(C, a_n) is omitted because its value is 1 as a side condition to the logic (otherwise the Update I rule would have to be used).

We can do the same for \( \text{size}_r(l) \), where \( l.a \) points to \( l_y \) instead of \( l_y \):

\[
\text{size}_r(l) = 1 + \sum_{i=1}^{n-1} \text{size}_r(v_i) \cdot B(C, a_i) + \sum_{j=n+1}^{k_y} \text{size}_r(v_j) \cdot B(C, a_j)
\]

If we now replace \( \text{size}_r(l) \) by \( \text{size}_r(l_y) \) by \( \text{size}_y \) and \( \text{size}_r(l_y) \) by \( \text{size}_y \), we see that \( \text{size}_r(l) = \text{size}_y \), so Valid(\( \eta, \tau, \{x : (C, p_x - p_a + p_y)\})

Aliasing Using Definition 5 we can determine that \( [l \cdot a]_\tau = l_y \). Then from \( \eta_x = l_x \), \( \eta_y = l_y \) and \( [l \cdot a]_\tau = l_y \), we can derive Valid(\( \eta, \tau, \{x \cdot a \rightarrow y\})

From OSUpdate we see that \( x \cdot a \rightarrow y \) evaluates to \( \eta_x \). Then from Definition 7 follows \( \eta_{\text{result}} = \eta_x = l_x \), so it is easy to see that Valid(\( \eta, \tau, \{\text{result} \rightarrow x\})

R’ About the sized types and aliasing relations between access paths \( r.a \) that are in \( R \) (which are unchanged in \( R’ \)) we know that \( \forall r.a \in R . [r.a]_\sigma \neq l \), so that \( \forall r.a \in R . [r.a]_\sigma = \tau \). We can now use Lemma 9 to derive that all aliasing in \( R \) is preserved: \( \forall b \sim r.a . a[\tau]_\sigma \rightarrow [r.a]_\sigma \).

When we look at part 2 of Definition 8 we see that for a sized type \( r.a : (C, p) \) in \( R’ \) to be valid it is required that \( r.a \) points to a proper heap location \( l \), this location contains an object of class \( C \) and size \( p \). Because Valid(\( \eta, \sigma, R \)), we already know that \( r.a \) points to a proper heap location \( l \), which contains a \( C \) object in heap \( \sigma \). Because \( \forall r.a \in R . \sigma[\tau]_\sigma = \tau[\tau]_\sigma \) and \( \forall r.a . [r.a]_\sigma = [r.a]_\sigma \), we know that \( \forall r.a \in R . \sigma[\tau]_\sigma = \tau[\tau]_\sigma \), i.e. that part \( R \) of the heap is constant around the field update (note the relation to separation logic). Thus, \( l_l \) indeed contains an object of class \( C \).

What is left is to prove that \( \text{size}(l) \) is indeed equal to the \( p \) specified in \( Q_{uf} \). Remember that in the construction of \( R’ \) from \( R \), the size \( p_{old} \) of an object \( r.a \) from which \( x \) can be reached (\( r.a \sim \rightarrow x \)) is adapted by subtracting \( \text{paths}(r, x) \) times the old size of \( x \) \( (p_x) \) and adding \( \text{paths}(r, x) \) times the new size of \( x \) \( (p_x - p_a + p_y) \).

So, we can distinguish two cases.

If \( l \sim \rightarrow l \) then \( p = p_{old} \). It follows immediately from Lemma 10 that \( \text{size}_r(l) = \text{size}_y \), so indeed \( p = p_{old} \).

If \( l \sim \rightarrow l \) then \( p = p_{old} - (p_x - p_a + p_y) \cdot \text{paths}(r, x) \). In the calculation of \( \text{size}_r(l) \) using Definition 4 we see that if \( v_i = l_y \) then \( \text{size}_r(v_i) = \text{size}_y (l_y) \) (instead of \( \text{size}_y (l_y) \) as in the calculation of \( \text{size}_r(l_y) \)), else \( \text{size}_r(v_i) = \text{size}_y (l_y) \).
size\_\sigma(v_i). The first case occurs exactly \(\text{paths}(r, x)\) times. So we can write 
\(\text{size}\_\tau(l_r)\) as 
\(\text{size}\_\sigma(l_r) - \text{size}\_\sigma(l_x) \cdot \text{paths}(r, x) + \text{size}\_\tau(l_x) \cdot \text{paths}(r, x)\). When we replace 
\(\text{size}\_\sigma(l_r)\) by \(p_{\text{old}}\), \(\text{size}\_\sigma(l_x)\) by \(p_x\) and \(\text{size}\_\tau(l_x)\) by \(p_x - p_a + p_y\) we see that 
\(p = p_{\text{old}} - p_x \cdot \text{paths}(r, x) + (p_x - p_a + p_y) \cdot \text{paths}(r, x)\).

**Done** We have now proved the validity of all the separate parts of \(Q_{uf}\), so from part 4 of Definition 8 it follows that \(\text{Valid}(\eta, \tau, Q_{uf})\).

### 5.3 Sketch of the Complete Proof

The proof for the entire strict-size logic should be done by induction on the size \(n\) of the derivation tree of the strict-size logic. For the case where \(n = 0\) all the axioms must be proved, including the derivation rules from the logic without productions above the line, like the UpdFrame rule. Then the inductive step, where \(n = k + 1\), is proved by case distinction of the strict-size logic rules.
Chapter 6

Evaluation and Future Work

In this chapter some suggestions for future work are given. The limitations of the current strict-size logic are described in Section 6.1, along with hints for possible solutions. In Section 6.2 a series of other possible extensions is described.

6.1 Limitations of the Program Logic

Although a lot of simple cases can be handled by the current logic, there are obvious limits to which code can be checked and what kinds of size relations can be expressed (and checked). These limitations are discussed in this section and hints for possible solutions are given.

6.1.1 Conditional Sizes

When sizes are changed within a conditional, only a common postcondition which can be derived from all branches of the conditional is certain. If in one branch, for instance, the size of $x$ is increased by one and in another branch it is decreased by two, it is unknown after the conditional.

One possible solution might be to say that in the postcondition of a conditional either $Q_1$ is true (the postcondition of the then-branch), or $Q_2$ is true (the postcondition of the else-branch in case $x \neq \text{null}$), or $Q_3$ is true (the postcondition of the else-branch in case $x = \text{null}$). This would however expand the proof exponentially, so it is not a very scalable solution.

Another solution, which is applied to functional size analysis in [42], is to collect dependencies and derive upper and lower bounds on the sizes.
6.1.2 Better Treatment of Aliasing

As for sized types, conditionals also form a problem for the aliasing relation. If the aliasing information is changed within a conditional, then after the conditional it is not sure which variables are aliases, i.e. some variables might be aliases, but might not.

In this work the aliasing problem is "solved" quite crudely. There is however an entire field of research with alias analysis as its main interest. For better treatment of aliasing the strict-size logic could be combined with an existing approach to alias analysis. A nice introduction of the field is given in [32]. Different methods of type-based alias analysis are compared in [17].

The general approach to the conditional aliasing problem is to have two types of aliases in the analysis: may-alias and must-alias. This would solve the conditional aliasing problem. However, how to deal with variables that may be aliases in the strict-size logic is big challenge. Also note that in [32] it is shown that the may-alias and must-alias problems are in general undecidable and uncomputable, respectively.

Methods that Affect Aliasing Between a Parameter and Other Variables

One common situation that cannot be handled by the current strict-size logic is when a method affects aliasing of a parameter in such a way that it may also influence aliasing between that parameter and other program variables if the method is invoked. An example of this is shown in Listing 6.1.

```plaintext
void newNext() {
    let List l = new List in
    let List _ = this.next <- l in
    return;
}
```

Listing 6.1: This method can affect aliasing of this.next

If the method above is called and this.next ← x for some object x, then this relation is broken. Such cases cannot be handled by the current logic. A crude solution to this problem would be to add another annotation which specifies that the aliasing between certain (fields of) variables may be broken. However, a thorough redesign of the current alias analysis in the form of a combination of the logic with an existing alias analysis would be a favorable extension.
6.1.3 Size is a Single Number

In this work the size of a data structure is given by a single integer. It can thus not be known from the size of a data structure how many objects of a particular class it contains.

For example consider a method replaceLeafs(Tree t) that replaces all the leafs of a tree with its parameter t. Say that the size of this tree is \( pt \) and that it has \( l \) leafs, and that the size of \( t \) is \( pt^2 \). Then the size of the tree after invocation of replaceLeafs(t) is invoked will be \( pt - l + l \cdot pt^2 \).

This is solved for a functional language by making the size of a data structure a tuple of the number of occurrences of each constructor in [45]. A similar solution could be applied here. However, there are still unsolved problems with this approach, for instance, because of polymorphism only the size of the “outer” data structure can be counted. This is similar to the non-counted elem fields in lists and trees in this work.

6.1.4 Other Restrictions

Besides the important limitations discussed in the previous sections, there are three smaller restrictions on programs for the logic. First, non-counted fields must be totally separated from counted fields, so that the sizes of counted field cannot depend on the non-counted ones. Second, the parameters of a method must be totally separated from each other, so that a change to one of the parameters cannot affect the others. And finally there may be no cyclic data.

Although no solutions are suggested these restrictions are listed here for completeness. These are basic restrictions in the sense that enable us to reason in a straightforward way about programs. If these restrictions were not assumed, then the strict-size logic would have become unnecessarily complex.

6.2 Other Extensions

Extensions that do not necessarily improve the current logic, but that would form valuable additions are discussed in this section.

6.2.1 Full Soundness Proof

In Chapter 5 a partial soundness proof is given. A useful extension would be to also prove the soundness of the remaining strict-size logic rules.
CHAPTER 6. EVALUATION AND FUTURE WORK

6.2.2 Implementation

A practical and useful extension of the work described herein would be an implementation. We expect that all the size analysis described herein can be automated. Such a program would involve a parser, type-checker and interpreter for the FJEU language, a size analysis engine, which implements the strict-size logic, and (an) algorithm(s) to check the aliasing annotations of methods, verify that no incomplete aliasing information is used and to derive the modifies sets needed for the frame rule.

A parser, type-checker and interpreter for RAJA is available from its website\footnote{http://www2.tcs.ifi.lmu.de/~rodrigue/raja.html}. A start has been made to modify this code into a parser, type-checker and interpreter for SAJA, which is available from my website\footnote{http://rodykersten.ruhosting.nl/fjeu.tgz}. So far, it implements a parser and interpreter for FJEU. Type-checking and parsing SAJA annotations are not implemented yet.

6.2.3 Size Inference

Currently, the annotations have to be given by the programmer. The addition of a method to automatically infer the annotations would yield a completely automatic system to derive size bounds. A size inference algorithm for polynomial size analysis on a functional language is described in [48]. This could be a good starting point to construct such an algorithm for deriving annotations as described herein.

6.2.4 Expansion to Java

Although FJEU is a simple variation to full Java that is very suitable for formal specifications and proofs such as those presented in this paper, it is not a language that is (intended to be) used in practice. For size analysis that is applicable in the “real world”, this research should be extended to Java. A possible route to achieve this is to extend size analysis to a subset of Java to which all Java programs can be transformed without affecting sizes (and prove that they can be transformed without affecting sizes).

\footnote{http://www2.tcs.ifi.lmu.de/~rodrigue/raja.html}
\footnote{http://rodykersten.ruhosting.nl/fjeu.tgz}
Chapter 7

Conclusions

In this work, a polynomial strict-size analysis for an imperative language has been developed. Although other research has been done on size and heap space analysis, this used functional languages and/or linear bounds, whereas the strict-size logic presented herein can be used to verify polynomial size relations for methods of an imperative language.

Most of the queues for this research come from the AHA project [47], in which polynomial size analysis is developed for a functional language. Another important line of research is the resource analysis by Hofmann and Jost [23, 31]. Their research also targets a functional language however and is limited to linear bounds. But, in contrast, their work has been developed much further in the sense that multiple resources can be analyzed (even at once), no annotations are needed and they have a working implementation which can handle complex test cases.

The main contribution of this work is the strict-size logic with which polynomial output-on-input size relations of methods in an imperative object-oriented language can be verified. More specifically, the frame rule, as described in Section 3.2.10, forms the kernel of our contribution. In a sense, it can be seen as both a specialization and a generalisation of the frame rule as used in separation logic [36, 40]. A specialization because, of course, the frame rule described herein is aimed at size analysis. A generalisation because apart from totally separated memory regions it can also handle cases where objects in the $P$ region are reachable from objects in the $R$ region by correctly adapting the information about $R$. Its soundness in conjunction with the field update rule is proved in Section 5.2 which shows that the separation concept is correct. A complete soundness proof for the strict-size logic is also sketched.

Although many more challenges lie ahead, the first steps in the development of polynomial size analysis for an imperative language have been taken. The initial difficulties of side-effects and aliasing are overcome successfully. Various suggestions for future work are given in Chapter 6 that are definitely worth investigating.
(importantly size-inference and an implementation of the program logic), so there-
fore hopefully this work can be an inspiration to other researchers and will be the
start of something larger!
Bibliography


BIBLIOGRAPHY


