Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

# Verificación Automática de Estructuras de Datos Acíclicas usando Demostradores de Teoremas

## Automatic Verification of Acyclic Data Structures using Theorem Provers

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Ariel Martín Neisen - L.U. 9/05

Directores: Dr. Diego Garbervetsky, Dr. Daniel Gorín

Buenos Aires, 2010

# Resumen

El objetivo de la presente tesis es investigar acerca del diseño de lenguajes orientados a objetos y las diferentes técnicas existentes para ofrecer garantías estáticas de verificación. En particular, nos interesa definir un *calificador de tipos acíclicos*, que asegure que la clausura transitiva de la relación *points-to* de una instancia acíclica sea irreflexiva. Listas enlazadas y árboles son típicos ejemplos de tipos acíclicos. Dicha propiedad es interesante debido a que: i) las estructuras de datos acíclicas pueden ser fácilmente recolectadas utilizando una estrategia de conteo de referencias (reference counting), y ii) es sencillo garantizar la terminación de ciclos que recorren estructuras de datos acíclicas. La discusión sobre aciclicidad nos llevará a entender cuán difícil es garantizarla con las herramientas disponibles en la actualidad.

Desde el punto de vista técnico, propusimos un lenguaje con un calificador de clases opcional "acíclico". El mismo impone algunas restricciones de tipado: si la clase A es declarada como acíclica y A contiene un campo "f" de tipo B, entonces B debe ser acíclico también. La aciclicidad es entonces forzada por construcción, o sea, agregando una precondición especial a la asignación "a.f := b", para "a" una instancia acíclica, que garantice la preservación de la aciclicidad de "a" y "b". La especificación es lograda utilizando una variación del trabajo realizado en *dynamic frames*. Uno de los problemas más interesantes a resolver es encontrar el nivel correcto de abstracción, en particular, cuál es la mínima información necesaria en el contrato de los métodos para que funcione correctamente la verificación. El trabajo sobre Dynamic Frames ofrece un enfoque elegante para resolverlo.

Las contribuciones de esta tesis incluyen la presentación de un nuevo lenguaje, con la definición formal de su semántica y las pruebas de su validez. Finalmente, se analizan experiencias realizadas que aprovechan los beneficios de los tipos acíclicos.

# Abstract

The aim of this thesis is to research on the design of object-oriented languages and the different techniques available to offer static verification guarantees. In particular, we became interested in defining *acyclic type qualifier*. By this we mean that the transitive closure of the *points-to* relation of an instance of an acyclic type must be irreflexive. Linked-lists and trees constitute typical examples of acyclic types. This is interesting because: i) acyclic data structures can be garbage collected automatically using a cheap reference counting strategy, and ii) loops that traverse acyclic data structures can be easily shown to be terminating. The discussion on acyclicity will lead us to understand how difficult it is to verify it using the current techniques available.

Technically speaking, we propose a language with an optional "acyclic" qualification to the classes declaration. This imposes some typing constraints: if class A is declared as acyclic and A contains a field "f" of type B, then B must have been declared as acyclic too. Acyclicity is then enforced by construction, that is, by adding a special precondition to the assignment "a.f := b" for "a" an instance of an acyclic type, that guarantees that the acyclicity of "a" and "b" are preserved. The specification is achieved by using a variation of the dynamic frames style. One interesting problem is to find the right level of abstraction: what is the minimum information needed to include in the contract of each method to make the verification work. The link with the specification of Dynamic Frames offers an elegant approach to help here.

The contributions of the work include the presentation of the new language, with the formal definition of its semantics and the proofs of soundness. We end up analyzing the experimental code samples, taking advantage of the acyclic types benefits.

# Agradecimientos

Este trabajo representa el fin de una etapa que comencé hace varios años. Durante ese tiempo mucha gente me ayudó de distintas formas y este es el momento de agradecerles (aunque posiblemente me olvide de alguno).

Conocí a Diego Garbervetsky y Dani Gorín en mis primeras materias cursadas en la carrera. Cuando llegó el momento de buscar un tema de tesis, no dudé en acudir a ellos como primera opción. Ellos confiaron en mi para llevar la tesis adelante y le dedicaron mucho esfuerzo y tiempo para que la pueda concluir de la mejor forma. Espero que este trabajo sea el inicio de una amistad por muchos años.

A Eduardo Bonelli y Guido de Caso por haber aceptado ser los jurados de la tesis. Sus correcciones, comentarios e ideas ayudaron a concluir con el desarrollo del trabajo. A Leo Spett y Pablo Zaidenvoren por haber sido revisores iniciales del documento.

A mis compañeros de la Facultad, la *gente de la facu*, con quienes compartí los últimos años dentro y fuera de la cursada: Matías Blanco, Fernando Bugni, Luis Brassara, Facundo Carreiro, Bruno Cuervo Parrino, Diego Freijo, Maxi Giusto, Pablo Laciana, Sergio Medina, Santiago Palladino, Leandro Radusky, Leo Rodriguez, Nati Rodriguez, Viviana Siless, Javier Silveira, Leo Spett, Andrés Taraciuk, Martín Verzilli, Pablo Zaidenvoren, Eddy Zoppi. Sin dudas que el apoyo del grupo fue fundamental para poder completar la carrera.

A mis compañeros de trabajo, por su aporte a mi desarrollo profesional.

Al Departamento de Computación de la Universidad de Buenos Aires por haberme dado una educación de calidad y permitido conocer personas brillantes. Al grupo de Ingeniería del Software, donde dí mis primeros pasos en la docencia.

A mis amigos Ari Brow, Fer Kahan, Mati Rubacha, Leo Spett, Nico Teitel y Zaiden, por ayudarme a despejar la cabeza y acompañarme en otro momento importante de mi vida.

A mi Familia, por su apoyo constante e incondicional, por alentarme a que me esfuerce en las cosas que me interesan y acompañarme en la construcción de mi camino.

# Contents

# Chapter 1

# Introduction

Writing correct and reliable programs is a very hard task. Whenever we get a piece of software, instead of coming with guarantees, it includes disclaimers and warnings. At the same time, software is becoming present in almost every aspect of our daily life. In consequence, a malfunctioning software may cause significant losses[Bro75].

Fortunately, both the academia and industry have made considerable progress in creating techniques, tools and methodologies which aim to improve the quality of software. In this thesis we follow the work in this area, with the intention of making a contribution to help programmers write better and more reliable software. Specifically, we are interested in analyzing programs. The analysis strategies can be divided into *static* (which take place before the program is executed) and *dynamic* (which take place during the execution of the program), or hybrid. There are many strategies, but for static analysis we will consider program verification and type systems, and for dynamic analysis we will consider testing.

Program Verification is one of the major research subjects of computer sciences. One of its applications is to prove that a program satisfies its specification. As the compiler has the role of generating the executable code from a program; a verifier is responsible to check if the program verifies its specification for all inputs and in each possible execution path. A program verifier usually works as follows:

- First of all, the program is formalized with regard to its semantics and proof obligations. A typical technique[Lei08a] is to translate the program into a verification language, which helps to prescribe the formalisms in a more natural way.
- Then, a module generates the logical formulas from the formalization (called *verification conditions*). The validity of the verification conditions implies that the program satisfies the correctness properties under consideration.
- Finally, the verification conditions are processed by a *theorem prover* searching for a successful proof or counterexamples that show possible errors in the program, such as the Z3[dMB08] satisfiability-modulo-theories (SMT) solver.

A type system is typically a tractable syntactic method for proving the absence of certain program undesired behaviors by classifying phrases to the kinds of values they compute[Pie02]. Type systems are a powerful (and nowadays natural) way of reasoning about programs and calculating a static approximation of its runtime behavior. They can be used to detect errors (the static type-checking allows early detection of programming errors) and improve abstraction, documentation (a typed program is more easily to read than a non-typed) and efficiency (some optimizations can be implemented using the type information).

Testing has shown to be a very powerful tool to find and correct bugs in large scale software development. Unit testing and different Extreme Programming[BA04] approaches are used in many industrial projects. Nevertheless, a very important amount of commercial software products reach the public with several bugs or the testing process takes a significant amount of resources and time. *Testing shows the presence, not the absence of bugs* (Edsger W. Dijkstra).

Certainly, program verification is still far from replacing manual testing. One of the techniques to accomplish it relies on the design-by-contract approach[Mey91]. It establishes that all software designers must define a formal and verifiable specification (also called contract) for the components that they are building. Besides from program verification, contracts are very useful to generate (and check) documentation. In Section 1.2 we will show the current state-of-the-art of program verification.

## 1.1 What this thesis is about

This thesis deals with the problem of verifying modular programs with the goal of enforcing some particular properties in runtime. We start researching on the language design and verification of modular object-oriented programs. This kind of programs have the benefit of scalability and the ability of changing a module implementation without any impact on the client. If we want to have a manageable verification process, we cannot afford to re-examine every module. That is why in modular program verification each module can be verified individually and then the specification is available for other client modules. Apart from being modular, we are also interested in verifying heap-manipulating programs; adding the challenge of finding the right level of abstraction to reason about them.

The property we have studied is acyclicity of objects. By this we mean that there will not be a cycle in runtime generated with the objects that it reaches. Linked-lists and trees constitute typical examples of acyclic data structures. The benefits of acyclic data structures are that:

1. Acyclic data structures can be garbage collected automatically using reference counting, which is a predictable garbage collection techniques. This property is very important in real time and embedded environments, where resource constraints require predictability of response times and available resources[CKP+08]. This idea is illustrated by the iPhone Memory Management[App09] (that uses the Objective-C language[Koc03]).

2. Loops that traverse acyclic data structures can be easily shown to terminate (if it is proven to progress). When analyzing the termination of a loop, if it iterates over an acyclic data structure, it will eventually reach the last element of the structure and terminate.

A garbage collection algorithm is in charge of keeping track of the memory used by a program and automatically releasing the locations that are not being used. Almost all the modern program execution environments rely on a garbage collection module. The algorithms can be divided into *reference counting* and *tracing*. A reference counting algorithm counts the amount of references that point to each object and when it gets to zero, the object can be deallocated. An execution system that implements a tracing algorithm takes a whole process to analyze the memory and release the locations that are unreachable. The reference counting approach is much more predictable than the tracing, since the memory analysis of the tracing can be executed at any time, putting at risk the execution time of the operations. However, reference counting has some disadvantages that limit its use in mainstream environments:

- The storage overhead that comes from keeping a count for each object.
- The execution overhead because of updating the reference count for each pointer operation.
- The inability to detect cycles (which is probably its greatest weakness). The reason for this is that a reference count of a cyclic object will always be greater than zero, and therefore it will never be deleted. It requires the programer to break cycles explicitly in the code or the use of a tracing algorithm to deal with cycles.

  Figure 1.1 illustrates the cyclic problem in reference counting. $r$ is one of the program root objects and the nodes in the gray area produce a cycle. In Figure 1.1a, $r$ reaches the cyclic nodes. When it does not reach them anymore (in Figure 1.1b), the gray area should be deallocated, because its objects are not reached by any of the program roots (in this case $r$). However, the reference count of those objects is greater than zero (since they are strongly connected) and will not be deallocated.

The contributions of this thesis are:


**(a)** Before **(b)** After

**Figure 1.1:** Memory garbage cycle example

- The formal definition of a language that enforces acyclicity
- The proof of soundness of the language
- A tool that statically checks the programs using a translation to Boogie[Lei08b]

## 1.2 Related Work

Program verification (as we know it) was first introduced by Floyd[Flo67] and Hoare[Hoa83]. They presented a formal system that consists of a set of axioms and inference rules which can be used in proofs of properties of computer programs. The central contributions of this system is the Hoare logic, which is based on triples and describes how the execution of a piece of code changes the state of the computation. The initial implementations were formalized from small procedure-oriented programming languages. Since those days, there's been plenty active work in the area trying to apply the program verification concepts into the modern object-oriented programming languages. This section provides a summary of the related work in the research of object-oriented programs verification.

Spec#[BLS04] is one of the most important works in verification of object-oriented programs. It is built as an extension to the popular C# language[HWG03]; supporting pre/post conditions, specification of abstractions, non-null types and loop invariants. It also includes support to the whole .NET Framework, in order to increase the adoption of the language. The verification is performed by translating the Spec# code into BoogiePL. We will go deep into BoogiePL translations in Chapter 5. Some of its ideas were taken for the Code Contracts support in C#4[Mic09a], which offers a design-by-contract programming methodology using static methods and decorations natively.

The Java Modeling Language (JML)[LBR99] is a behavioral interface specification language designed to specify Java modules. It allows to add explicit annotations for the module's clients. JML is also being used by specific purpose tools for verification and memory consumption.

The Extended Static Checking for Java (ESC/Java)[FLL+02] is a tool that tries to find common programming errors statically. It uses code annotations and tries to find inconsistencies between the design and the actual code implementation. ESC/Java design tries to trade-off soundness and usefulness to reduce the annotations cost and to improve performance. In practice, users have complained that the amount of annotations needed is heavy and that it throws too many false warnings.

The Java Type Annotations Specification[Ern08] is an extension that allows annotations to appear in almost all the uses of a type. Those annotations can be written in unusual locations, such as generic types arguments. One of its benefits is that it is planned to be part of the Java 7 language, allowing a native support for annotations. Then, any type-checking tool can detect errors using the information provided by the programer.

There are some works that try to deal with the acyclicity problem. Shape Analysis[SRW02] concerns the problem of determining invariants of programs that manipulate dynamically allocated storage. Using the shape graph, it might be possible to analyse the acyclicity. Another approach is to use memory regions[LP04]. Even though this approach has shown its high points, it introduces a programming style for that purpose. In our work, we try to solve the acyclicity problem using the style that is used in a standard object-oriented verification language.

Another interesting approach to make static verification in the memory graph is using Ownership types[CPN98]. This provides a flexible way of restricting the visibility of object references and relations, enforcing object encapsulation statically. It introduces the concepts of owner (which controls the access to an object) and a representation (the objects owned by an object). The idea is that an object can own subobjects it depends on, preventing them to be accessible from the outside. Ownership can also guarantee acyclicity by enforcing a tree data structure. This is a more restrictive approach than the one that we will present in this thesis, because we allow any kind of acyclic data structures.

In Section 2.2 we present the related work regarding the Dynamic Frames style, which is a very interesting approach to modular program specification. We will present the motivation and solution, including the experimental languages that have been implemented by different authors.

## 1.3 Thesis Structure

Chapter 2 introduces the framing problem when specifying and verifying object-oriented programs and how to deal with it. With that goal in mind, we describe the dynamic frames approach from its motivations, up to the prototypes that illustrate the concept.

Chapter 3 goes deep into understanding what it means when we say that a reference is acyclic and how difficult it is to verify it. We will see that acyclicity cannot be expressed with a first-order logic formula and, therefore, should be guaranteed by construction instead.

Chapter 4 presents the simple language that we will use throughout this thesis. We will present the syntax (which is similar to the one of a simple Java-like language), the interesting features that we are supporting and the dynamic and static semantics. Then we define the formalism for verifying the execution properties that we are interested in and we prove that it guarantees the acyclicity of references.

Chapter 5 takes the formalism defined in Chapter 4 and the related work in program verification in order to put into practice our language. It starts with the background work in which we base the implementation and we go over all the details of the translations from our language into the verification one. The chapter ends with the experiments we have done and the analysis of its results.

Finally, Chapter 6 presents the conclusions and contributions of this thesis, and discusses what areas of future work it opens.

# Chapter 2

# Dynamic Frames

When verifying modular programs, one of the critical aspects is how to specify the area of the memory that a method can access, which is called the *framing problem*[LLM07]. This chapter presents an introduction of one of the alternatives to deal with that problem and the related work in the subject. Most of the approaches are based on the *dynamic frames* style introduced by Ioannis Kassios [Kas08, Kas06].

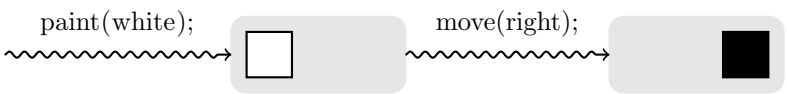
## 2.1 Overview

Abstraction[LG86] is one of the central concepts when designing and programming object-oriented programs. This means that the details about how a class (or method) is implemented can be suppressed, and an implementation can be replaced by any other respecting the same expected behavior without affecting the overall result. One possible solution is to create an abstraction that exposes the method's specification that separates the way it is implemented internally. The same notion applies to the way in which a class is structured with regards to its internal data (which is called data abstraction). Data abstraction is a methodology that enables us to isolate the data that a class exposes from how it is internally constructed. Therefore, programs should operate over *abstract data*, without making assumptions on how each component is implemented [AS96].

However, one of the obstacles of applying abstraction is the *framing problem*[LLM07]. A method's frame expresses what it is allowed to change during its execution, in other words, the part of the state that it operates upon. Without it, a specification will not be very useful. Let's take a look at an example of this problem in an abstract manner[1]. Suppose we have a square shape with two operations:

- paint(color), with the following contract: "After the method call, the square will be painted according to the color given as a parameter"
- move(dir), with the following contract: "After the method call, the square will be moved according to the direction given as a parameter"

Then consider the execution sequence "paint(white);move(right)" illustrated in the following figure:




The final outcome was not the expected, right? The square's color should be white, not black. The problem is that from move(right) we can only conclude that the square will be moved to the right, but it does not say anything about the color preservation.

[1] Example taken from http://pm.ethz.ch/teaching/ws2006/SemSpecVer/slides/Dynamic_Frames.pdf

The same problem occurs when verifying a program. Kassios solution for the framing problem consists of using *dynamic frames*[Kas06]: a specification variable that represents a set of memory locations and is used to specify the effect of methods in an abstract matter. A *specification variable* is an abstract representation of the state that is exposed to the client, but is not taken into account at runtime. Dynamic Frames provides an interesting style to describe classes in which an object is implemented in terms of another (will be shown with some examples in the rest of the section). It brought a more powerful style to specify modular classes, and still is amenable to use in the current verifiers.

We will continue the explanation using some code examples.

```
class Cell {
    var value : Integer;
    var footprint : Set<Object>;

    Cell()
        ensures getValue() == null;
        ensures footprint == {};
    {
        value := null;
        footprint := {};
    }

    method getValue()
      returns (r : Integer)
        ensures r == value;
    {
        r := value;
    }

    method setValue(Integer newValue)
        modifies footprint;
        ensures getValue() == newValue;
        ensures footprint == {newValue};
    {
        value := newValue;
        footprint := { value };
    }
}
```

**(a)** The Cell class

```
method main() {
    Cell c1 = new Cell();
    c1.setValue(1);

    Cell c2 = new Cell();
    c2.setValue(2);

    assert c1.getValue() == 1;
}
```

**(b)** Client program

**Figure 2.1:** A simple dynamic frames specified class with its client program

Figure 2.1a shows a simple Cell class source code. The *Cell* class has an explicit footprint field which is a set of memory locations that specifies its frame. The *setValue* method modifies what *value* references. However, if we explicitly specify it, we will expose Cell's internal representation. Therefore, using the footprint it maintains the information hiding and allows the programer to specify the contracts in an implementation independent way. The **modifies** clause of the contract frames the objects on which the method is allowed to have an effect, apart from allocating and modifying new objects. If the modifies expression is a footprint, it means that the method can change the object that it includes. On the other hand, if the **modifies** clause is not defined, the method is side-effects free.

Let's informally follow the execution of the client program from Figure 2.1b. The first statement (Cell c1 = **new** Cell()) does not modify any living object (since the modifies clause is not defined) and ensures that c1's footprint is empty. Then, the c1.setValue(1) statement only modifies the locations in c1 pre state and ensures that c1.getValue() is 1. Following that, it executes an allocation (Cell c2 = **new** Cell()) that creates a new empty footprint for c2. c2's footprint is disjoint from c1's because the allocation statement creates an object that was not allocated before the execution of the object

(also called *fresh* object). That is why the next statement c2.setValue(2) does not affect c1.footprint (and, in consequence, c1.getValue()), concluding that **assert** c1.getValue() == 1 will be valid.

```
class Node {
    var footprint : set<Node>;
    var next : Node;

    function Valid() : bool
    {
        this in footprint &&
        (next != null ⇒ next ∈ footprint &&
            (next.footprint ⊆ footprint) &&
            next.Valid()
        )
    }

    method Init()
        modifies this;
        ensures Valid();
    {
        next := null;
        footprint := {this};
    }

    method Prepend(first: Node)
        requires Valid() && first.next == null;
        ensures first.Valid();
        ensures first.footprint == { first } ∪ footprint
    {
        first.next := this;
        first.footprint := { first } ∪ footprint;
    }
}
```

**Figure 2.2:** Node class representing a linked-list. It requires the first .getNext() == **null** for simplicity in the explanation.

Up to this point we have not seen clearly the *dynamic* property of the dynamic frames. The Node class in Figure 2.2 represents a linked-list with a prepend method that connects the parameter node with the list. The *Valid* function works as an explicit class invariant that enforces the state of the footprint (the footprint of a node must include the footprint of all the nodes found following the *next* field). Then, when executing prepend(first), first 's footprint grows because it is connected to the rest of the list. In other words, prepend(first)'s frame change dynamically during the execution of the program. Notice that the programer must update the footprint manually, both in the specification and method body. This has the disadvantages of making it more difficult to write a program and it relies on the programmer to produce a correct specification. A part of our work (mostly in Section 4) consists of automating the operations over the footprints.

## 2.2 Implementations based on Dynamic Frames

This section presents the different works related to implementing a language following the dynamic frames style in different scenarios.

Dafny[Lei08a] is an experimental language created by K. Rustan M. Leino that explores the dynamic

frames style of specifications in an object-based sequential setting. Both Figures 2.1 and 2.2 are written in Dafny language. Some of its features are:

- It is a very simple object-oriented language (similar to Java) that supports method specification using dynamic frames.
- Needs to explicitly specify the effects on the footprint
- Supports complex predicates (like the Valid function in Figure 2.2).

This work is described in Section 5.1.3 since the implementation we proposed is build upon Dafny's base.

SJava[SJPS08] is a small Java-like language that does not support features such as exceptions and multithreading. One interesting aspect of this work is that it includes inheritance. It extends regular Java so that it supports contract specification and specification variables for dynamic frames. The verifier is build based on the Spec# program verifier and it uses the Z3 and Simplify theorem provers[dMB08].

Another work presented the notion of Implicit Dynamic Frames[SJP09]. It proposes a variant of the dynamic frames style inspired by separation logic[PB08]. This approach eliminates the need to explicitly write and check frame annotations, like in the other implementations. The solution consists of specifying in a contract the accessibility to the memory locations that a method is intended to read or write. This work shows a significant difference with regards to the others because it separates completely the implementation from the specification of the effects in the memory.

Chalice[LM09, LMS09] is a language and program verifier that detects bugs in concurrent programs by verifying the absence of data races and deadlocks. The implementation is built on top of Implicit Dynamic Frames to specify access of a method over a memory location using fractional permissions[Boy03]. This is an example of how to leverage the concept of dynamic frames to solve other problems. VeriCool[SJP08] implements a similar approach.

# Chapter 3

# Dealing with acyclicity in an Object-Oriented language

In Chapter 1 we discussed why we are interested in guaranteeing acyclic references. This chapter studies how to enforce this property.

Given a programming language that supports contract specification, we need to provide means to enforce acyclicity. One way is to extend its type system, axioms and formal specification to statically verify that all *acyclic annotated* references are acyclic. In the code of Figure 3.1 we present a simple Node class with some client methods. Look at its declaration: **acyclic class** Node. The *acyclic* qualifier defines the class to be acyclic. The link method connects its first Node parameter with the second (n1.setNext(n2)). Then, the last line of the testCycle method invokes the link method, which would create a cycle between someNode and someOtherNode. That statement should throw an error, since it breaks the acyclicity of both objects. This does not happen in testAcyclic, where the final Node structure is acyclic.

```
acyclic class Node
{
    private Node next;

    public Node getNext()
        { return next; }

    public void setNext(n : Node)
        { next = n; }
}

method link(n1 : Node, n2 : Node)
{
    // . . .
    // some code
    // . . .

    n1.setNext(n2);
}
```

```
method testCycle()
{
    Node someNode = new Node;
    Node someOtherNode = new Node;

    link(someNode, someOtherNode);
    link(someOtherNode, someNode);
}

method testAcyclic()
{
    Node someNode = new Node;
    Node someOtherNode = new Node;
    Node finalNode = new Node;

    link(someNode, someOtherNode);
    link(someOtherNode, finalNode);
}
```

**Figure 3.1:** *testCycle* shows a program that produces a cycle and *testAcyclic* produces an acyclic list

Our first idea to solve this problems was to think of the acyclicity as a class invariant. A class invariant[HK00] expresses which states of an object of the class are consistent or *legal*. When updating an object, we need to be sure that its invariant holds. In practice, a method requires that the invariant

holds as a precondition and ensures it when returning. In our case, we can generate an *isAcyclic* predicate for each class expressing that it is acyclic. Then, all the method's specifications would implicitly include **this**.isAcyclic() for all the references in the requires and ensures clauses. Moreover, this approach would enable to break the acyclicity inside a method body and reestablish it before returning. Figure 3.2 presents this idea. In line 6 a cycle is generated between node and its successor. Then, before returning from the method (in line 12) the acyclicity is reestablished, verifying the ensures clause node.isAcyclic().

```
method reestablishingAcyclicity(node : Node)
    requires node.isAcyclic();
    ensures node.isAcyclic();
{
    node.setNext(new Node());
6:  node.getNext().setNext(node);

    // . . .
    // more method code
    // . . .

12: node.getNext().setNext(null);
}
```

**Figure 3.2:** Breaking the acyclicity and restoring it before the method returns

Despite the fact that the invariant approach seems natural and would be the classic approach using C#[BLS04] or JML[LBR99], we came across with a difficulty. The acyclicity predicate is not expressible in first order logic! In consequence, approaches based in SMT solvers would not work (since we cannot write the acyclicity invariant for the classes). Let's show why this happens. First of all, we need to recall the Compactness Theorem.

**Compactness Theorem.** *Let* $\Gamma$ *be any set of formulas of first-order logic. Then:*

- *If* $\forall\ \Gamma_0$ *finite,* $\Gamma_0 \subseteq\ \Gamma, \Gamma_0$ *is unsatisfiable* $\Rightarrow\ \Gamma$ *is unsatisfiable.*

We want to prove that there does not exist a formula $IsAcyclic(x)$, with a free variable $x$, so that $\mathcal{M} \models IsAcyclic[v] \Leftrightarrow$ not exists $a_1, ..., a_n \in [\mathcal{M}]$ such that $a_1 = v(x)$ and $\mathcal{R}^{\mathcal{M}}(a_n, a_j)$ and $\mathcal{R}^{\mathcal{M}}(a_1, a_2)$ and ... and $\mathcal{R}^{\mathcal{M}}(a_{n-1}, a_n)$ for any $j \in \{1, ..., n\}$. $\mathcal{R}$ is a symbol that represents a binary relation, in our case a reference reachability.

Lets assume that $IsAcyclic(x)$ is expressible. In that case, the formula $\neg IsAcyclic(x)$ is true in a model iff exists a finite cycle from $x$. That is, for some $i$, $Acyclic_i$ is true, where $i \in \{1, ..., n\}$.

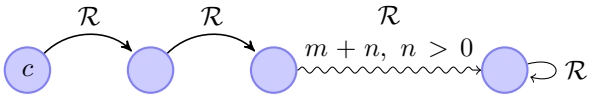
$$
\begin{aligned}
Acyclic_0(x) &\equiv \neg\mathcal{R}(x, x) \quad \text{(there is no cycle of length 0)} \\
Acyclic_1(x) &\equiv \neg(\exists y_1, y_2)(x = y_1 \wedge \mathcal{R}(y_1, y_2) \wedge y_2 = y_1) \\
Acyclic_2(x) &\equiv \neg(\exists y_1, y_2, y_3)(x = y_1 \wedge \mathcal{R}(y_1, y_2) \wedge \mathcal{R}(y_2, y_3) \wedge (y_3 = y_1 \vee y_3 = y_2) \\
&\vdots \\
Acyclic_n(x) &\equiv \neg(\exists y_1, y_2, ..., y_n)(x = y_1 \wedge \mathcal{R}(y_1, y_2) \wedge \mathcal{R}(y_2, y_3) \wedge ... \wedge \mathcal{R}(y_{n-1}, y_n) \\
&\qquad \wedge (y_n = y_1 \vee y_n = y_2 \vee ... \vee y_n = y_{n-1})
\end{aligned}
$$

Let $c$ be a constant. Consider $\Gamma = \{\neg IsAcyclic(c), Acyclic_0(c), Acyclic_1(c), ...\}$.

**Fact 1** $\Gamma$ is unsatisfiable. If it were satisfiable then, for some model $\mathcal{M}$, $\mathcal{M} \models \Gamma$. In particular, $\mathcal{M} \models \neg IsAcyclic(c)$, which implies that
$c^{\mathcal{M}}$has a cycle of length $\kappa$, and therefore $\mathcal{M} \not\models IsAcyclic_\kappa(c)$, but $IsAcyclic_\kappa(c) \in \Gamma$

**Fact 2** $\Gamma$ is satisfiable. Let $\Gamma_0$ be finite so that $\Gamma_0 \subseteq \Gamma$. Therefore, for some $m$ $\Gamma_0 \subseteq \{\neg IsAcyclic(c), Acyclic_1(c), Acyclic_2(c), ..., Acyclic_m(c)\} = \Gamma_1$

Let's show that $\Gamma_1$ is satisfiable. Take any model $\mathcal{M}$ where $c^{\mathcal{M}}$ is cyclic in more than $m$ steps.




Since $\Gamma_0 \subseteq \Gamma_1, \Gamma_0$ is satisfiable too. By Compactness, $\Gamma$ is satisfiable.

From **Fact 1** and **Fact 2** we get a contradiction, which comes from assuming that IsAcyclic is expressible in first-order logic.

In conclusion, we were not able to implement our first approach. Our second idea was to enforce acyclicity by construction, that is guaranteeing that after the execution of each statement, the acyclic objects are acyclic. This imposes some constrains on the programs that we will support, for instance the one from Figure 3.2 will fail, no matter if the acyclicity is reestablished at the end of the method, since the acyclic objects should always be acyclic.

## 3.1 Enforcing acyclicity by construction

Type qualifiers[FFA99] encode a simple but highly useful form of subtyping. Since it is a declarative way to add properties to a type, we implemented an *acyclic type qualifier* to identify the classes whose objects should be acyclic.

Then we chose to enforce acyclicity by construction. One of the decisions we made is how to reason about the memory. In chapter 2 we presented how Dynamic Frames approaches the framing problem by introducing a specification variable whose value is a set of allocations. [Lei08a] and [SJPS08] refer to that variable as *footprint*, implementing and maintaining it in different ways. The footprint of a method specifies an upper bound of the memory locations that it reads or writes.

We took the footprint concept and specialized it to allow finer grained specifications about the state of the memory. Instead of using just a set, we implement footprints as a multiset of references that specify the memory locations that can be reached from any given reference, in particular from how many simple paths[1] it can be reached.

**Definition 1.** *Given $ref_1, ref_2$, $ref_1.footprint[ref_2]$ returns the number of different simple paths from $ref_1$ to $ref_2$ (where the paths are defined by the object's fields)*

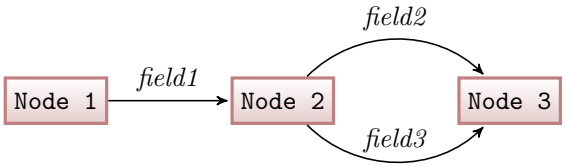



**Figure 3.3:** An acyclic example

The footprint of an object $a$ with respect to another object $b$ counts how many ways $a$ can reach to $b$. In Figure 3.3 we can see that:

- $node1.footprint[node1] == 1$ since the reference is only reachable from itself
- $node1.footprint[node3] == 2$ since $node3$ can only be reached from $node1$ through $node2$. $node2$ reaches $node3$ from both $field2$ and $field3$, so its $footprint$ value is 2

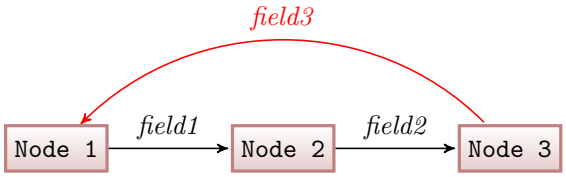



**Figure 3.4:** A cyclic example

In Figure 3.4 we can see that:

- $node3$ generates a cycle because its connection to $node1$
- $node1.footprint[node1] == 2$ since the reference can also be reached from $node1 \rightarrow node2 \rightarrow node3 \rightarrow node1$

It is worth mentioning that when a reference's footprint value for itself it is greater that 1, we are in the presence of a cycle. This concept will be used later in Chapter 4 when we will present the semantics of our language.

**Definition 2.** *An Acyclic Type Qualifier defines a type such that:*

1. *All its fields are acyclic qualified*
2. *Verifies the allocation, assignment and invariant conditions defined in as follows:*

### Allocation Conditions

When a new object of an acyclic type is allocated, we need to initialize its footprint in order to ensure that the object is fresh. Figure 3.5 illustrate how the memory changes after the allocation is executed.

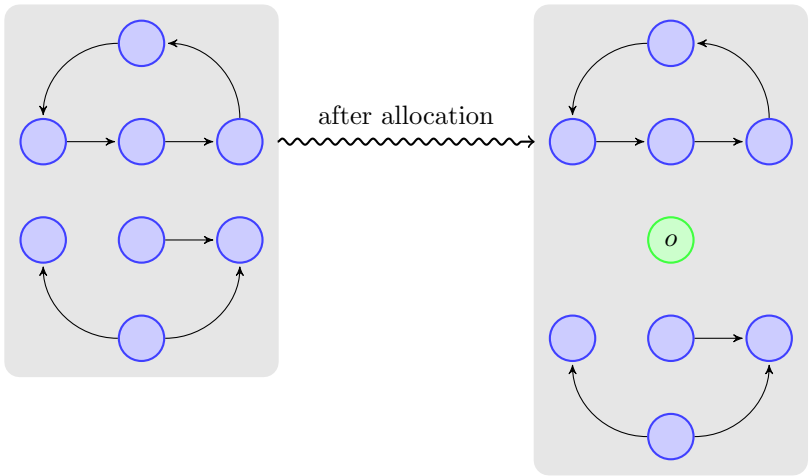



**Figure 3.5:** Memory changes after allocation

On the left there is the heap memory with some objects already allocated and on the right we can see the effects of allocating an object ($o$ is a reference to it). Notice that:

- The previously allocated objects do not reach $o$

[1] Counting simple paths works fine with acyclic data structures. However, in the presence of cycles we need to take some special considerations that are left for future work.

- $o$ can only reach itself

More formally, the effect on the footprint after the allocation of a new object (referenced by $o$) is[2]:

$$(\forall\ r : Ref)(r \neq o \Rightarrow o.footprint[r] = 0) \wedge$$
$$(\forall\ r : Ref)(r \neq o \Rightarrow r.footprint[o] = 0) \wedge$$
$$o.footprint[o] = 1$$

### Assignments Conditions

When we write a field, we need to take some precautions, since the operation may create a cycle. That is why before executing an *acyclic* assignment "$o.f = d$", we need to require that the target object ($d$) does not reach the receiving one ($o$). Figure 3.6 shows an acyclic memory state and how an assignment generates a cycle.

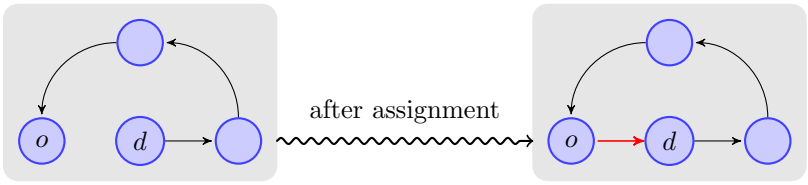



**Figure 3.6:** An assignment that generates a cycle

In order to prevent this case, before an assignment we need to require that:

$$d.footprint[o] = 0$$

In order to verify that the assignments do not generate cycles, we need to correctly model the effect of those assignments in the footprint. Potentially, the footprint of all references may be affected, so we need to provide a granular specification. Figure 3.7 illustrates the effect on the memory after executing "$o.f = d$". Notice that not only $o$ ends up reaching $d$, but also $a$ and $b$ (since those objects also reach $o$).

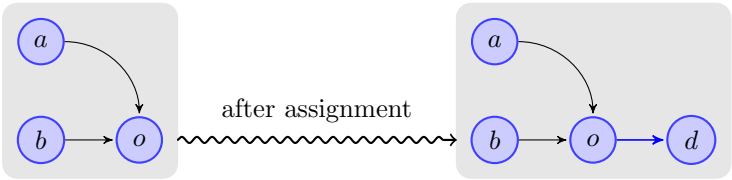



**Figure 3.7:** Memory changes after a field assignment

More formally, the changes in the footprint after "$o.f = d$" are:

- If $o.f$ was previously assigned, it will be removed from the footprint of objects that reach $o$

$$((old(o).f \neq null \wedge d = null) \Rightarrow ((\forall\ r_1, r_2 : Ref)(r_1.footprint[o] > 0 \Rightarrow$$
$$r_1.footprint[r_2] = old(r_1).footprint[r_2] - old(o).f.footprint[r_2])))$$

- Not only $d$, but also all the objects that $d$ reaches are added to the footprint of the objects that reach $o$

$$((old(o).f = null \wedge d \neq null) \Rightarrow ((\forall\ r_1, r_2 : Ref)(r_1.footprint[o] > 0 \Rightarrow$$
$$r_1.footprint[r_2] = old(r_1).footprint[r_2] + d.footprint[r_2])))$$

[2]This predicate does not express that the footprints where $o$ is not present have the same value as before the allocation execution. The complete formal definition will be presented in Chapter 4

- If $o.f$ was previously assigned and $d$ is not null, the effect is defined using the former cases

$$\begin{aligned}((old(o).f \neq null \wedge d \neq null) \Rightarrow ((\forall\ r_1, r_2 : Ref)(r_1.footprint[o] > 0 \Rightarrow \\ r_1.footprint[r_2] = \\ old(r_1).footprint[r_2] - old(o).f.footprint[r_2] + d.footprint[r_2])))\end{aligned}$$

**Footprint invariant for acyclic qualified types**

Each acyclic qualified type has an implicit invariant that characterizes its basic structure in the heap, in terms of the reachability of the living objects. An object footprint is the specification variable that represents that structure. Therefore, the footprint invariant should check:

1. The footprints do not include *null*, since it does not point to any object
2. The reference reaches itself only once, otherwise it will present a cycle (illustrated in Figure 3.4)
3. For each (non-null) field in the reference type:
   - The reference reaches the field at least once
   - The field's footprint is a subset of the reference's footprint
   - The field preserves the footprint invariant

More formally, the footprint invariant can be considered as the following axiom:

$$\begin{aligned}(\forall o : Ref)(FootprintInvariant(o) \Leftrightarrow\ & (o.footprint[o] = 1 \wedge \\ & o.footprint[null] = 0 \wedge \\ & (\forall\ field : Field) \\ & \quad (o.footprint[o.field] \geq 1 \wedge \\ & \quad o.field.footprint \subseteq o.footprint \wedge \\ & \quad FootprintInvariant(o.field))))\end{aligned}$$

It is worth mentioning that in this last predicate there is an universal quantification over the fields. As a matter of fact, this predicate can be auto-generated for each class because the field members are fixed in compile time in our language.

# Chapter 4

# A language that enforces acyclicity

In Chapter 3 we discussed the difficulties of expressing acyclicity in a first order formula using a SMT solver and the proposed solution: enforcing acyclicity by construction.

This chapter formally presents a language that supports acyclic type qualifiers. We will show a comprehensive definition, including the language grammar, the first order logic (for the contract specifications) and the semantics.

We should also take into consideration that as our programs get larger, the amount of specification predicates needed in the method's contract becomes difficult. For that, the semantics definition enforces the constraints in order to avoid the explicit declaration of acyclicity conditions in the contracts. We also include macros for some predicates.

## 4.1 Syntax

The syntax of the language that will be used throughout this section is defined in Figure 4.1. It is a simple object-oriented language, with the following properties:

- Provides the level of Object-Orientation that can be found in a limited Java language (we are not supporting inheritance and polymorphism)
- Includes contract specifications based on *Dynamic Frames*
- Supports the acyclic type qualifier that enables the programer to specify an acyclic type.
- Defines an implicit *footprint* member in all classes. Recalling Section 3.1, we have decided to express the reachable objects using a multiset footprint. Some other experiments[Lei10] have shown that this level of granularity provides a flexible and simple specification.

For the sake of simplicity, we are not supporting primitive types (such as integers or booleans). In case of needing those types, the programer can define them as explicit classes. This will help us to present more clear formal definitions and proofs; without losing expressibility (the programer can implement an integer class).

We are not going to define in detail the type system (which is similar to Dafny's[Lei08a]) and it is assumed that all the programs used when defining the semantics type successfully.

The expression of the conditional is a reference, instead of using a boolean. If the reference is non-null, the true statements are executed. Otherwise, the execution continues with the false case.

We do not implement the while control flow statement in order to simplify the semantics. However, it can be implemented using recursion and conditionals.

Methods include contracts with pre and post conditions predicates, like in Eiffel[Mey92], JML[LBR99] and Spec#[BLS04]. If a method contract cannot be proven, the programer will get an error from the

verifier. We have defined a first-order logic to express those predicates, which is shown in Figure 4.2. The precondition ($Expr_{requires}$) refers to the execution state before the method invocation using the free variable $\mathcal{S}_{pre}$ and the postcondition ($Expr_{ensures}$) can refer to the state after and before it using the free variables $\mathcal{S}_{pre}$ and $\mathcal{S}_{pos}$ respectively (for example, it may use the value of a field before the method was executed).

As we have discussed in Chapter 2, an important part of a method contract is to specify what parts of the state it modifies (the *framing* problem). This part is expressed in the modifies clause. We allow a method to modify a single object or all the objects included in a multiset.

$$
\begin{array}{rcl}
Program & ::= & Class^* \\
Class & ::= & \textbf{acyclic class } ClassName\ \{Member^*\}\ | \\
 & & \textbf{class } ClassName\ \{Member^*\} \\
Member & ::= & Field \mid Method \\
Field & ::= & \textbf{var } id : ClassName \\
Method & ::= & \textbf{method } Id(Param^*)\ \textbf{returns}\ (Param)\ Spec^*\ \{\ Stmt\ \} \\
Param & ::= & id : ClassName \\
Stmt & ::= & Expr := Expr \\
 & | & id := new\ Type \\
 & | & \textbf{if } (Expr)\{Stmt\}\ \textbf{else } \{Stmt\} \\
 & | & id := Expr.Id(Expr^*) \\
 & | & Stmt; Stmt \\
 & | & \textbf{skip} \\
Expr & ::= & id \mid \textbf{this} \mid \textbf{null} \mid id.f \mid \textbf{this}.f \\
Expr_{mod} & ::= & id \mid \textbf{this} \mid id.footprint \mid \textbf{this}.footprint \\
Spec & ::= & \textbf{requires } Expr_{requires}; \mid \textbf{modifies } Expr_{mod}{}^*; \mid \textbf{ensures } Expr_{ensures};
\end{array}
$$

**Figure 4.1:** Our programming language. The expressions of the **requires** and **ensures** clauses are the formulas presented in Figure 4.2.

To understand the grammar and method's specification, in Figure 4.3 we show a typical linked-list acyclic data structure. In that program the reader can find that:

- The *setFirst* method precondition requires that *node* does not reach the list, because otherwise it would create a cycle after the execution.
- The *setFirst* method postcondition states that **this**.$first == node$. Notice that we do not specify the effect on the resulting footprint, because of the footprint invariant (definition 2).

## 4.2 Dynamic Operational Semantics

The presentation of the semantics is inspired in [DVE00]. First of all we will introduce the program state in Figure 4.4.

The smallest elements of the state are objects and references. An object represents an instance of a class $C$ and is noted $\ll f_1 : \iota_1, ..., f_n : \iota_n \gg^{\mathtt{C}}$. $f_i$ are the fields defined in the class $C$. The footprint is a special field (named `Footprint`). Each field value holds a reference. A reference (named $\iota, \iota_i, \kappa, \kappa_i$) is a link to an object and **null** is a special reference that does not point to any object.

The *Store* (named $\sigma$) is a partial mapping of references to objects. The mapping function is injective (i.e., $ref_1 \neq ref_2 \Rightarrow \sigma(ref_1) \neq \sigma(ref_2)$).

**Definition 3.** *Given an object* $\mathtt{obj} = \ll \mathtt{f_1} : \iota_1, ..., \mathtt{f_n} : \iota_n \gg^{\mathtt{C}}$ *of class* `C` *and a store* $\sigma$*, the operations over the objects and stores are defined as:*

**Sorts**

$Store \mid Ref \mid Num \mid Field$

**Terms**

| | |
|---|---|
| $\text{storeRead}(t_{\mathcal{S}}, t_r, t_f) : Ref$ | $t_{\mathcal{S}}$ is $Store$, $t_r$ is $Ref$, $t_f$ is $Field$ |
| $\bar{0} : Num$ | |
| $\bar{1} : Num$ | |
| $t_1 \dot{+} t_2 : Num$ | $t_1, t_2$ are $Num$ |
| $t_1 \dot{-} t_2 : Num$ | $t_1, t_2$ are $Num$ |
| $footprint(t_{\mathcal{S}}, t_r, t'_r) : Num$ | $t_{\mathcal{S}}$ is $Store$, $t_r$ is $Ref$, $t'_r$ is $Ref$ |
| $\mathbf{null} : Ref$ | |
| $\mathbf{this} : Ref$ | |

**Formulas**

| | |
|---|---|
| $alive(t_r, t_{\mathcal{S}})$ | $t_r$ is a $Ref$, $t_{\mathcal{S}}$ is a Store |
| $isAcyclicQual(t_r)$ | $t_r$ is a $Ref$ |
| $t_r == t'_r \mid t_r \neq t'_r$ | $t_r$ and $t'_r$ are Ref |
| $t_1 == t_2 \mid t_1 \neq t_2 \mid t_1 < t_2 \mid t_1 > t_2$ | $t_1$ and $t_2$ are Num |
| $\neg F \mid F_1 \wedge F_2$ | $F,\ F_1, F_2$ are formulas |
| $\forall x \bullet F$ | $x$ is a $Ref$, $F$ is a formula |
| $\forall t_{\mathcal{S}} \bullet F$ | $t_{\mathcal{S}}$ is a $Store$, $F$ is a formula |

**Macros**

$\mathcal{S}(o.f) \equiv storeRead(\mathcal{S}, o, f)$

$new(\mathcal{S}_{pre}, \mathcal{S}_{pos}, x) \equiv \neg alive(x, \mathcal{S}_{pre}) \wedge alive(x, \mathcal{S}_{pos})$

**Figure 4.2:** Formation rules of the terms and formulas of the first order logic used to specify the method's contracts. $\mathcal{S}(o.f)$ is a macro for the *storeRead* term and returns the value of $o.f$. *alive* returns whether the instance is alive in the current store (if the parameter is **null**, the function returns **true**). $new(\mathcal{S}_{pre}, \mathcal{S}_{pos}, x)$ is a macro for $\neg alive(x, \mathcal{S}_{pre}) \wedge alive(x, \mathcal{S}_{pos})$ and returns whether the reference is a new fresh object. *isAcyclicQual* returns if the reference's type is acyclic qualified.

```
acyclic class Node {
    var next : Node;
}

acyclic class List {
    var first : Node;

    method setFirst(node : Node)
        requires node ≠ null ∧ storeRead(S_pre, node, next) == null;
        // Prevent a cycle between this and node
        requires footprint(S_pre, node, this) == 0;
        // The method's frame is the footprint of this
        modifies this.footprint;
        // Specify the expected effect of the method
        ensures storeRead(S_pos, this, first) == node ∧
            storeRead(S_pos, node, next) ==
                storeRead(S_pre, this, first);
    {
        node.next := first;
        this.first := node;
    }
}
```

**Figure 4.3:** Code sample showing the use of the grammar

$$
\begin{array}{rcl}
\mathit{Store} & ::= & \mathit{Ref} \rightarrow \mathit{Object} \\
\mathit{Object} & ::= & \ll (\mathit{Field} : \mathit{Ref}\ )^* \gg^{\texttt{ClassName}} \\
\mathit{Ref} & ::= & \iota \mid \textbf{null} \\
\mathit{Env} & ::= & \texttt{id} \rightarrow \mathit{Ref}
\end{array}
$$

**Figure 4.4:** Program state

$$\mathtt{obj}(\mathtt{f_i}) = \iota_\mathtt{i}$$

$$\mathtt{obj}[\mathtt{f} \mapsto \kappa](\mathtt{f}') = \begin{cases} \kappa & \text{if } \mathtt{f}' = \mathtt{f} \\ \mathtt{obj}(\mathtt{f}') & \text{otherwise} \end{cases}$$

$$\sigma(\iota, \mathtt{f}) = \sigma(\iota)(\mathtt{f})$$

$$\sigma[\iota \mapsto \mathtt{z}](\iota') = \begin{cases} \mathtt{z} & \text{if } \iota' = \iota \\ \sigma(\iota') & \text{otherwise} \end{cases}$$

$$\sigma[\iota, \mathtt{f} \mapsto \kappa] = \sigma[\iota \mapsto \sigma(\iota)[\mathtt{f} \mapsto \kappa]]$$

$$typeof(\ll ... \gg^C) = C$$

*Env* is a partial mapping of local variables (or identifiers) to references. The identifiers will be named $v, v_i$.

The program state is defined with a 3-tuple $< \sigma, env, \textbf{this} >$, where $\sigma$ is a store, *env* is the environment and **this** is a reference; such that $Img(env) \setminus \{\textbf{this}\} \subseteq dom(\sigma)$. In other words, each variable in the environment has to be associated with an object in the store.

The program's semantics is modeled using the following runtime mappings:

$$\begin{array}{rcl} Configuration_{stmt} & ::= & Stmt \times (Store \times Env \times Ref) \\ Configuration_{expr} & ::= & Expr \times (Store \times Env \times Ref) \\ \rightsquigarrow & : & Configuration_{stmt} \rightarrow (Store \times Env) \\ \overset{e}{\rightsquigarrow} & : & Configuration_{expr} \rightarrow Ref \end{array}$$

$Configuration_{stmt}$ is a tuple of statements (the ones defined in Figure 4.1), stores, environments and the context **this** reference[1]. Likewise, $Configuration_{expr}$ defines the semantics of the expressions execution. The expression will be named $e, e_i$.

A big step semantics (i.e., the one that describes the final outcome of the execution of a statement) is better suited for our purposes, because it helps to analyze this semantics with regards to the static one (see Section 4.5).

We will present the dynamic semantics using an iterative approach: we begin with a reduced Java language and refine its semantics incorporating dynamic frames first and acyclicity constrains later. Figures 4.5 and 4.6 define the semantics of the first basic language.

Most of the rules of this section are quite standard. Rule D_CALL of Figure 4.6 defines how to execute a method call ($v := e_0.\mathtt{m}(e_1, ..., e_n)$). Since we implement a big step semantics, first we execute the statements of the method's body with the appropriate *this* reference, the initial $\sigma$ store and a new environment that maps the parameter names to the references of the invocation. After that, the outcome of the method call is the resulting $\sigma'$ store and the initial environment with the $v$ variable mapping to the value of the *ret* variable of the method body. Notice that we are not performing any runtime check with the pre or post conditions. We will define how to check if a method contract fails in Section 4.3.

### 4.2.1 Dynamic Frames language dynamic semantics

Following the discussion on Chapter 2, here we extend the language to ensure that a method invocation does not modify any object beyond its frame. Rule D_FIELD_W_2 updates the footprints after writing an

[1] We do not include the reference to **this** in the outcome because it does not change after the execution of the statement

$$\text{D\_V}\ \frac{env(v) = \iota}{v, \sigma, env, this \overset{e}{\rightsquigarrow} \iota} \qquad \text{D\_FIELD\_R}\ \frac{\mathtt{e}, \sigma, env, this \overset{e}{\rightsquigarrow} \iota \quad \iota \text{ is not } \mathbf{null}}{\mathtt{e.f}, \sigma, env, this \overset{e}{\rightsquigarrow} \sigma(\iota, \mathtt{f})}$$

$$\text{D\_THIS}\ \frac{}{\mathbf{this}, \sigma, env, this \overset{e}{\rightsquigarrow} this} \qquad \text{D\_NULL}\ \frac{}{\mathbf{null}, \sigma, env, this \overset{e}{\rightsquigarrow} \mathbf{null}}$$

**Figure 4.5:** Evaluation of expressions

$$\text{D\_SEQ}\ \frac{\mathtt{stmt_1}, \sigma, env, this \rightsquigarrow \sigma', env' \quad \mathtt{stmt_2}, \sigma', env', this \rightsquigarrow \sigma'', env''}{\mathtt{stmt_1; stmt_2}, \sigma, env, this \rightsquigarrow \sigma'', env''}$$

$$\text{D\_IF\_THEN}\ \frac{e, \sigma, env, this \overset{e}{\rightsquigarrow} \iota \quad \iota \textit{ is not } \mathbf{null} \quad \mathtt{stmt_1}, \sigma, env, this \rightsquigarrow \sigma', env'}{\mathbf{if}\ (e)\ \{\mathtt{stmt_1}\}\ \mathbf{else}\ \{\mathtt{stmt_2}\}, \sigma, env, this \rightsquigarrow \sigma', env'}$$

$$\text{D\_IF\_ELSE}\ \frac{e, \sigma, env, this \overset{e}{\rightsquigarrow} \mathbf{null} \quad \mathtt{stmt_2}, \sigma, env, this \rightsquigarrow \sigma', env'}{\mathbf{if}\ (e)\ \{\mathtt{stmt_1}\}\ \mathbf{else}\ \{\mathtt{stmt_2}\}, \sigma, env, this \rightsquigarrow \sigma', env'}$$

$$\text{D\_SKIP}\ \frac{}{\mathbf{skip}, \sigma, env, this \rightsquigarrow \sigma, env}$$

$$\text{D\_NEW}\ \frac{\iota \text{ is fresh} \quad \sigma' = \sigma[\iota \mapsto\ll \mathtt{f_1} : \mathbf{null}, ..., \mathtt{f_n} : \mathbf{null} \gg^{\mathtt{C}}]}{v := \mathbf{new}\ \mathtt{C}(), \sigma, env, this \rightsquigarrow \sigma', env[v \mapsto \iota]}$$

$$\text{D\_FIELD\_W}\ \frac{o, \sigma, env, this \overset{e}{\rightsquigarrow} \iota \quad v, \sigma, env, this \overset{e}{\rightsquigarrow} \kappa \quad \iota \text{ is not } \mathbf{null}}{o.f := v, \sigma, env, this \rightsquigarrow \sigma[\iota, f \mapsto \kappa], env}$$

$$\text{D\_VAR\_W}\ \frac{e, \sigma, env, this \overset{e}{\rightsquigarrow} \iota}{v := e, \sigma, env, this \rightsquigarrow \sigma, env[v \mapsto \iota]}$$

$$\text{D\_CALL}\ \frac{\begin{array}{l} e_i, \sigma, env, this \overset{e}{\rightsquigarrow} \iota_i \\ \iota_0 \text{ is not } \mathbf{null} \text{ and } typeof(\sigma(\iota_0)) = \mathtt{C} \\ \mathtt{C.m} = \mathbf{method}\ m(p_1 : T_1, ..., p_n : T_n)\ \mathbf{returns}\ (ret : T_{ret}) \\ \quad \mathbf{requires}\ Pre;\ \mathbf{modifies}\ Mod;\ \mathbf{ensures}\ Post; \\ \quad \{\ Body\ \} \\ env' = [p_1 \mapsto \iota_1, ..., p_n \mapsto \iota_n, ret \mapsto \mathbf{null}] \\ Body, \sigma, env', \iota_0 \rightsquigarrow \sigma', env'' \end{array}}{v := e_0.\mathtt{m}(e_1, ..., e_n), \sigma, env, this \rightsquigarrow \sigma', env[v \mapsto env''(ret)]}$$

**Figure 4.6:** Semantics of the statements execution

object field, because we must add the paths that the new reference reaches and remove the ones from the old value. Figure 3.7 illustrates this case. In rule D_NEW_2 we only initialize the footprint of the new object, explained in Figure 3.5. Finally, in Rule D_CALL_2 we need to check if the method execution violates its frame. First we execute the expressions of the **modifies** clause and then we verify that the objects either have the same value as before executing the method or belong to the frame.

D_FIELD_W_2

$$\frac{\begin{array}{l} o, \sigma, env, this \overset{e}{\rightsquigarrow} \iota \\ v, \sigma, env, this \overset{e}{\rightsquigarrow} \kappa \\ \iota \text{ is not } \textbf{null} \\ \sigma' = \sigma[\iota, f \mapsto \kappa] \\ \sigma(\iota, f) \neq \textbf{null} \text{ and } \kappa \neq \textbf{null} \text{ implies that for all } \iota_1,\ \sigma(\iota_1, \texttt{Footprint})[\iota] > 0 \\ \quad \text{implies that } \sigma'(\iota_1, \texttt{Footprint}) = \sigma(\iota_1, \texttt{Footprint}) - \sigma(\iota, f)(\texttt{Footprint}) + \\ \quad\quad \sigma(\kappa, \texttt{Footprint}) \\ \sigma(\iota, f) \neq \textbf{null} \text{ and } \kappa = \textbf{null} \text{ implies that for all } \iota_1,\ \sigma(\iota_1, \texttt{Footprint})[\iota] > 0 \\ \quad \text{implies that } \sigma'(\iota_1, \texttt{Footprint}) = \sigma(\iota_1, \texttt{Footprint}) - \sigma(\iota, f)(\texttt{Footprint}) \\ \sigma(\iota, f) = \textbf{null} \text{ and } \kappa \neq \textbf{null} \text{ implies that for all } \iota_1,\ \sigma(\iota_1, \texttt{Footprint})[\iota] > 0 \\ \quad \text{implies that } \sigma'(\iota_1, \texttt{Footprint}) = \sigma(\iota_1, \texttt{Footprint}) + \sigma(\kappa, \texttt{Footprint}) \end{array}}{o.f := v, \sigma, env, this \rightsquigarrow \sigma', env}$$

D_NEW_2

$$\frac{\begin{array}{l} \iota \text{ is fresh} \\ \sigma' = \sigma[\iota \mapsto\ll \texttt{f}_1 : \textbf{null}, ..., \texttt{f}_n : \textbf{null}, \texttt{Footprint} : \{\iota : 1\} \gg^{\texttt{C}}] \end{array}}{id := \textbf{new } \texttt{C}(), \sigma, env, this \rightsquigarrow \sigma', env[id \mapsto \iota]}$$

D_CALL_2

$$\frac{\begin{array}{l} e_i, \sigma, env, this \overset{e}{\rightsquigarrow} \iota_i \\ \iota_0 \text{ is not } \textbf{null} \text{ and } typeof(\sigma(\iota_0)) = \texttt{C} \\ \texttt{C.m} = \textbf{method } m(p_1 : T_1, ..., p_n : T_n) \textbf{ returns } (ret : T_{ret}) \\ \quad \textbf{requires } Pre; \textbf{ modifies } Mods; \textbf{ ensures } Post; \\ \quad \{\ Body\ \} \\ env' = [p_1 \mapsto \iota_1, ..., p_n \mapsto \iota_n, ret \mapsto \textbf{null}] \\ Body, \sigma, env', \iota_0 \rightsquigarrow \sigma', env'' \\ \text{For all } e_{mod,i} \text{ in } Mods,\ e_{mod,i}, \sigma, env', \iota_0 \overset{e}{\rightsquigarrow} \iota_{mod,i} \\ CheckFrame(\sigma, \sigma', \iota_{mod,0}, ..., \iota_{mod,n}) \end{array}}{v{:=}e_0.\texttt{m}(e_1, ..., e_n), \sigma, env, this \rightsquigarrow \sigma', env[v \mapsto env''(ret)]}$$

$$CheckFrame(\sigma, \sigma', \iota_{mod,0}, ..., \iota_{mod,n}) \equiv$$
$$\text{for all } \iota,\ \sigma(\iota) = \sigma'(\iota) \text{ or } InMod(\iota, \iota_{mod,0}) \text{ or ... or } InMod(\iota, \iota_{mod,n})$$

$$InMod(\iota, \iota_{mod,i}) \equiv \begin{cases} \iota = \iota_{mod,i} & \text{if } \iota_{mod,i} \text{ is Ref} \\ \iota \in \iota_{mod,i} & \text{if } \iota_{mod,i} \text{ is Multiset} \end{cases}$$

The rule D_FIELD_W_2 would be very difficult to implement if we were designing the compiler for this language. The complexity of the statement would be $O(|\sigma|)$, because in worst case all the objects in the store might be affected. We will discuss if this operation can be optimized in the static semantics Definition 9.

### 4.2.2 Acyclicity-aware language dynamic semantics

The last step in the presentation of the semantics modifies Rule D_FIELD_W_2 in order to prevent the generation of cycles in an acyclic object. In Chapter 3 we analyzed that before executing a field assignment, we need to check that it will not create a cycle (illustrated in Figure 3.6). For that reason, Rule D_FIELD_W_3 fails the execution of the assignment ($o.f := v$) if $o$ is included in $v$'s footprint (which would produce a cycle). Figure 4.7 extends the definition of the semantics, introducing Rule D_FIELD_W_3.

$$\text{D\_FIELD\_W\_3}\quad \frac{\begin{array}{l} o, \sigma, env, this \overset{e}{\rightsquigarrow} \iota \\ v, \sigma, env, this \overset{e}{\rightsquigarrow} \kappa \\ \iota \text{ is not } \textbf{null} \\ \sigma(\kappa, \texttt{Footprint})[\iota] = 0 \\ \sigma' = \sigma[\iota, f \mapsto \kappa] \\ \sigma(\iota, f) \neq \textbf{null} \text{ and } \kappa \neq \textbf{null} \text{ implies that for all } \iota_1,\ \sigma(\iota_1, \texttt{Footprint})[\iota] > 0 \\ \quad \text{implies that } \sigma'(\iota_1, \texttt{Footprint}) = \sigma(\iota_1, \texttt{Footprint}) - \sigma(\iota, f)(\texttt{Footprint}) + \\ \quad\quad \sigma(\kappa, \texttt{Footprint}) \\ \sigma(\iota, f) \neq \textbf{null} \text{ and } \kappa = \textbf{null} \text{ implies that for all } \iota_1,\ \sigma(\iota_1, \texttt{Footprint})[\iota] > 0 \\ \quad \text{implies that } \sigma'(\iota_1, \texttt{Footprint}) = \sigma(\iota_1, \texttt{Footprint}) - \sigma(\iota, f)(\texttt{Footprint}) \\ \sigma(\iota, f) = \textbf{null} \text{ and } \kappa \neq \textbf{null} \text{ implies that for all } \iota_1,\ \sigma(\iota_1, \texttt{Footprint})[\iota] > 0 \\ \quad \text{implies that } \sigma'(\iota_1, \texttt{Footprint}) = \sigma(\iota_1, \texttt{Footprint}) + \sigma(\kappa, \texttt{Footprint}) \end{array}}{o.f := v, \sigma, env, this \rightsquigarrow \sigma', env}$$

**Figure 4.7:** Semantics of an allocation to prevent cycles

## 4.3 Failing Executions

While the operational semantics defined in the previous section represents the executions of programs and the impact on the state, we are also interested in ensuring some properties in the state after the operational execution. In order to prove those properties, we need to distinguish the executions that *fail* from the ones that do not terminate. By failure we mean if occurs an assertion failure or if the dynamic semantics gets stuck (because we use big step semantics, the latter includes non-termination). For that reason, we are introducing in this section the definition of the fails predicate:

$$\mathsf{fails} \quad :: \quad Store \times Env \times Ref \times (Stmt \cup Expr)$$

When the fails predicate holds, it represents that the *Stmt* statement or *Expr* expression has failed. Like in the previous one, this Section explains the definition of the function for the different *kind* of languages. Before giving the definitions of the fails predicate, we need to explain how to *make sure* that a method contract is valid (or not). Section 4.1 defines the details of the language syntax, specially the first order language that expresses the predicates. Therefore, we need to define an interpretation of the first-order language with regards to a dynamic state.

**Definition 4.** *The following set of definitions present the interpretation of the language from Figure 4.2.*

*The valuation of the variables is the following:*

$v_r : Vars_{Ref} \rightarrow Ref$

$v_s : Vars_{Store} \rightarrow Store$

*The interpretation of the language is:*

***Terms of sort Store***

$\mathcal{I}^{Store}_{v_s, v_r, this}(x_{\mathcal{S}}) = v_s[x_{\mathcal{S}}]$

***Terms of sort Ref***

$\mathcal{I}^{Ref}_{v_s, v_r, this}(x_r) = v_r[x_r]$

$\mathcal{I}^{Ref}_{v_s, v_r, this}(\textbf{null}) = \textbf{null}$

$\mathcal{I}^{Ref}_{v_s, v_r, this}(\textbf{this}) = this$

$\mathcal{I}^{Ref}_{v_s, v_r, this}(storeRead(t_{\mathcal{S}}, t_r, t_f)) = \mathcal{I}^{Store}_{v_s, v_r, this}(t_{\mathcal{S}})(\mathcal{I}^{Ref}_{v_s, v_r, this}(t_r), \mathcal{I}^{Field}_{v_s, v_r, this}(t_f))$

***Terms of sort Num***

$$\mathcal{I}^{Num}_{v_s,v_r,this}(\bar{0}) = 0$$

$$\mathcal{I}^{Num}_{v_s,v_r,this}(\bar{1}) = 1$$

$$\mathcal{I}^{Num}_{v_s,v_r,this}(x_1 \dot{+} x_1) = \mathcal{I}^{Num}_{v_s,v_r,this}(x_1) + \mathcal{I}^{Num}_{v_s,v_r,this}(x_2)$$

$$\mathcal{I}^{Num}_{v_s,v_r,this}(x_1 \dot{-} x_1) = \mathcal{I}^{Num}_{v_s,v_r,this}(x_1) - \mathcal{I}^{Num}_{v_s,v_r,this}(x_2)$$

$$\mathcal{I}^{Num}_{v_s,v_r,this}(footprint(t_{\mathcal{S}}, t_r, t'_r)) = \mathcal{I}^{Store}_{v_s,v_r,this}(t_{\mathcal{S}})(\mathcal{I}^{Ref}_{v_s,v_r,this}(t_r), \texttt{Footprint})[\mathcal{I}^{Ref}_{v_s,v_r,this}(t'_r)]$$

***Terms of sort Field***

$$\mathcal{I}^{Field}_{v_s,v_r,this}(x_f) = x_f$$

***Formulas***

$$v_s, v_r, this \models alive(t_r, t_{\mathcal{S}}) \Leftrightarrow \mathcal{I}^{Ref}_{v_s,v_r,this}(t_r) \textit{ is not } \textbf{null} \textit{ implies that is in the domain of } \mathcal{I}^{Store}_{v_s,v_r,this}(t_{\mathcal{S}})$$

$$v_s, v_r, this \models isAcyclicQual(t_r) \Leftrightarrow \mathcal{I}^{Ref}_{v_s,v_r,this}(t_r) \textit{ type is annotated as acyclic}$$

$$v_s, v_r, this \models t_r == t'_r \Leftrightarrow \mathcal{I}^{Ref}_{v_s,v_r,this}(t_r) = \mathcal{I}^{Ref}_{v_s,v_r,this}(t'_r)$$

$$v_s, v_r, this \models t_r \neq t'_r \Leftrightarrow \mathcal{I}^{Ref}_{v_s,v_r,this}(t_r) \neq \mathcal{I}^{Ref}_{v_s,v_r,this}(t'_r)$$

$$v_s, v_r, this \models t_1 == t_2 \Leftrightarrow \mathcal{I}^{Num}_{v_s,v_r,this}(t_1) = \mathcal{I}^{Num}_{v_s,v_r,this}(t_2)$$

$$v_s, v_r, this \models t_1 \neq t_2 \Leftrightarrow \mathcal{I}^{Num}_{v_s,v_r,this}(t_1) \neq \mathcal{I}^{Num}_{v_s,v_r,this}(t_2)$$

$$v_s, v_r, this \models t_1 < t_2 \Leftrightarrow \mathcal{I}^{Num}_{v_s,v_r,this}(t_1) < \mathcal{I}^{Num}_{v_s,v_r,this}(t_2)$$

$$v_s, v_r, this \models t_1 > t_2 \Leftrightarrow \mathcal{I}^{Num}_{v_s,v_r,this}(t_1) > \mathcal{I}^{Num}_{v_s,v_r,this}(t_2)$$

$$v_s, v_r, this \models \neg F \Leftrightarrow v_s, v_r, this \not\models F$$

$$v_s, v_r, this \models F_1 \wedge F_2 \Leftrightarrow v_s, v_r, this \models F_1 \text{ and } v_s, v_r, this \models F_2$$

$$v_s, v_r, this \models \forall x \bullet F \Leftrightarrow v_s, v_r[x \mapsto r], this \models F \text{ for all } r \in Ref$$

$$v_s, v_r, this \models \forall \mathcal{S} \bullet F \Leftrightarrow v_s[\mathcal{S} \mapsto s], v_r, this \models F \text{ for all } s \in State$$

*As a syntactic sugar, we allow the following interpretation expressions:*

$$\begin{array}{lcl} \sigma, v_r, this \models \varphi & \equiv & \{\mathcal{S}_{pre} \mapsto \sigma\}, v_r, this \models \varphi \\ \{\sigma, \sigma'\}, v_r, this \models \varphi & \equiv & \{\mathcal{S}_{pre} \mapsto \sigma, \mathcal{S}_{pos} \mapsto \sigma'\}, v_r, this \models \varphi \end{array}$$

The rules for the definition of the fails predicate are close to the dynamic semantics ones. For the executions of statements in a basic Java language, a failing execution is one in which a method contract is violated or when we try to access an invalid object (like a **null** reference, a null reference in a method invocation or an undeclared variable). We also need to make sure that the *execution failure* is correctly propagated over the flow of the different statements. All those definitions can be found on Figure 4.8.

Rule F_CALL_1 holds when the initial state does not satisfy the contract precondition. On the other hand, Rule F_CALL_2 holds when the resulting state does not satisfy the postcondition. And finally, Rule F_CALL_3 holds when the statements of the method's body fail.

When we include dynamic frames support in the language, the *modifies* clause adds some constraints that must be checked after the invocation of a method: only the references in that clause can be modified and the footprint should be *valid*. Taking that into account, in Rule F_CALL_4 from Figure 4.9 we extend the definition of the fails predicate in order to hold:

$$\text{F\_SEQ\_1}\quad \frac{\mathsf{fails}(\sigma, env, this, stmt_1)}{\mathsf{fails}(\sigma, env, this, stmt_1; stmt_2)} \qquad \text{F\_SEQ\_2}\quad \frac{stmt_1, \sigma, env, this \rightsquigarrow \sigma', env' \quad \mathsf{fails}(\sigma', env', this, stmt_2)}{\mathsf{fails}(\sigma, env, this, stmt_1; stmt_2)}$$

$$\text{F\_IF\_1}\quad \frac{\mathtt{e}, \sigma, env, this \overset{e}{\rightsquigarrow} \iota \quad \iota \textit{ is not } \textbf{null} \quad \mathsf{fails}(\sigma, env, this, \mathtt{stmt_1})}{\mathsf{fails}(\sigma, env, \textbf{if } (\mathtt{e}) \ \{\mathtt{stmt_1}\} \textbf{ else } \{\mathtt{stmt_2}\})}$$

$$\text{F\_IF\_2}\quad \frac{\mathtt{e}, \sigma, env, this \overset{e}{\rightsquigarrow} \iota \quad \iota \textit{ is } \textbf{null} \quad \mathsf{fails}(\sigma, env, this, \mathtt{stmt_2})}{\mathsf{fails}(\sigma, env, \textbf{if } (\mathtt{e}) \ \{\mathtt{stmt_1}\} \textbf{ else } \{\mathtt{stmt_2}\})}$$

$$\text{F\_NULL}\quad \frac{o, \sigma, env, this \overset{e}{\rightsquigarrow} \iota \quad \iota \text{ is } \textbf{null}}{\mathsf{fails}(\sigma, env, this, o.f)} \qquad \text{F\_CALL\_NULL}\quad \frac{id \text{ is an identifier} \quad e_i, \sigma, env, this \overset{e}{\rightsquigarrow} \iota_i \quad \iota_0 \text{ is } \textbf{null}}{\mathsf{fails}(\sigma, env, this, id := e_0.\mathtt{m}(e_1, ..., e_n))}$$

$$\text{F\_CALL\_1}\quad \frac{\begin{array}{l} e_i, \sigma, env, this \overset{e}{\rightsquigarrow} \iota_i \\ \iota_0 \text{ is not } \textbf{null} \text{ and } typeof(\sigma(\iota_0)) = \mathtt{C} \\ id \text{ is an identifier} \\ \mathtt{C.m} = \textbf{method } m(p_1 : T_1, ..., p_n : T_n) \textbf{ returns } (ret : T_{ret}) \\ \quad \textbf{requires } Pre; \textbf{ ensures } Post; \\ \quad \{ \ Body \ \} \\ env' = [p_1 \mapsto \iota_1, ..., p_n \mapsto \iota_n, ret \mapsto \textbf{null}] \\ \sigma, env', \iota_0 \not\models Pre \end{array}}{\mathsf{fails}(\sigma, env, this, id := e_0.\mathtt{m}(e_1, ..., e_n))}$$

$$\text{F\_CALL\_2}\quad \frac{\begin{array}{l} e_i, \sigma, env, this \overset{e}{\rightsquigarrow} \iota_i \\ \iota_0 \text{ is not } \textbf{null} \text{ and } typeof(\sigma(\iota_0)) = \mathtt{C} \\ id \text{ is an identifier} \\ \mathtt{C.m} = \textbf{method } m(p_1 : T_1, ..., p_n : T_n) \textbf{ returns } (ret : T_{ret}) \\ \quad \textbf{requires } Pre; \textbf{ ensures } Post; \\ \quad \{ \ Body \ \} \\ id := e_0.\mathtt{m}(e_1, ..., e_n), \sigma, env, this \rightsquigarrow \sigma', env' \\ \{\sigma, \sigma'\}, env', \iota_0 \not\models Post \end{array}}{\mathsf{fails}(\sigma, env, this, id := e_0.\mathtt{m}(e_1, ..., e_n))}$$

$$\text{F\_CALL\_3}\quad \frac{\begin{array}{l} e_i, \sigma, env, this \overset{e}{\rightsquigarrow} \iota_i \\ \iota_0 \text{ is not } \textbf{null} \text{ and } typeof(\sigma(\iota_0)) = \mathtt{C} \\ id \text{ is an identifier} \\ \mathtt{C.m} = \textbf{method } m(p_1 : T_1, ..., p_n : T_n) \textbf{ returns } (ret : T_{ret}) \\ \quad \textbf{requires } Pre; \textbf{ ensures } Post; \\ \quad \{ \ Body \ \} \\ env' = [p_1 \mapsto \iota_1, ..., p_n \mapsto \iota_n, ret \mapsto \textbf{null}] \\ \mathsf{fails}(\sigma, env', \iota_0, Body) \end{array}}{\mathsf{fails}(\sigma, env, this, id := e_0.\mathtt{m}(e_1, ..., e_n))}$$

**Figure 4.8:** Definition of the fails predicate for a basic Java language

- if a non-modifiable object changes after the invocation, or
- if the footprint state is invalid. Those conditions over the footprint are needed to enforce acyclicity by construction (discussed in Section 3.1).

F_CALL_4

$$\frac{\begin{array}{l} e_i, \sigma, env, this \overset{e}{\rightsquigarrow} \iota_i \\ \iota_0 \text{ is not } \textbf{null} \text{ and } typeof(\sigma(\iota_0)) = \texttt{C} \\ id \text{ is an identifier} \\ \texttt{C.m} = \textbf{method } m(p_1 : T_1, ..., p_n : T_n) \textbf{ returns } (ret : T_{ret}) \\ \quad \textbf{requires } Pre; \textbf{ modifies } Mods; \textbf{ ensures } Post; \\ \quad \{ \ Body \ \} \\ id := e_0.\texttt{m}(e_1, ..., e_n), \sigma, env, this \rightsquigarrow \sigma', env' \\ \text{For all } e_{mod,i} \text{ in } Mods, \ e_{mod,i}, \sigma, env', \iota_0 \overset{e}{\rightsquigarrow} \iota_{mod,i} \\ CheckFrame(\sigma, \sigma', \iota_{mod,0}, ..., \iota_{mod,n}) \text{ does not hold} \\ \sigma, env \not\models FootprintInvariant \text{ or } \sigma', env' \not\models FootprintInvariant \end{array}}{\mathsf{fails}(\sigma, env, this, id := e_0.\texttt{m}(e_1, ..., e_n))}$$

$\forall \sigma, \iota \bullet FootprintInvariant(\sigma, \iota) \Leftrightarrow (isAcyclicQual(\iota)$ implies that
$\sigma(\iota, \texttt{Footprint})[\iota] = 1)$ and
$(\neg isAcyclicQual(\iota)$ implies that $\sigma(\iota, \texttt{Footprint})[\iota] \geq 1)$ and
(for all $f$ field in $\iota, \sigma(\iota, \texttt{Footprint})[\sigma(\iota, f)] \geq 1$ and
(for all $\iota_1$ $Ref$, $\sigma(\iota, \texttt{Footprint})[\iota_1] \geq \sigma(\sigma(\iota, f), \texttt{Footprint})[\iota_1]$ and
$FootprintInvariant(\sigma, \sigma(\iota, f))))$

**Figure 4.9:** Definition of the fails predicate after a method invocation in a Dynamic Frames language

To finish the definition of the fails predicate, we need to add a rule that holds when a field assignment produces a cycle. In Figure 3.6 we have shown how this statement may violate the acyclicity invariant. The Rule F_FIELD_W from Figure 4.10 completes the presentation of the rules of the fails predicate. In Section 4.5 we will prove the soundness of the language and its semantics; showing the reasons behind the definitions of this section.

F_FIELD_W

$$\frac{\begin{array}{c} o, \sigma, env, this \overset{e}{\rightsquigarrow} \iota \\ v, \sigma, env, this \overset{e}{\rightsquigarrow} \kappa \\ \kappa \neq \textbf{null} \text{ and } isAcyclicQual(\iota) \text{ and } \sigma(\kappa, \texttt{Footprint})[\iota] > 0 \end{array}}{\mathsf{fails}(\sigma, env, this, o.f := v)}$$

**Figure 4.10:** Definition of the fails predicate for a field assignment in an Acyclicity Aware language.

## 4.4 Static Semantics

This section presents a Hoare-style logic for our language, allowing to define the static semantics which will be used to verify the programs implementation. A very interesting work on this subject can be found in [PHM99]. The first order language that we will be referring was defined in Figure 4.2.

The way to express each statement and expression is in the form of { **P** } *Stmt* { **Q** } triples, where **P**, **Q** are first order formulas that can be considered to be the pre and post conditions, respectively. The **P** formulas have as free variable the $\mathcal{S}_{pre}$ store and the **Q** has the $\mathcal{S}_{pre}$ and $\mathcal{S}_{pos}$ stores (since the postconditions can make reference to values in the pre-state of the execution). Just as syntactic sugar, in **P** $o.f$ is equivalent to $\mathcal{S}_{pre}(o.f)$; and in **Q** $o.f$ is equivalent to $\mathcal{S}_{pos}(o.f)$ and $pre(o.f)$ is equivalent to $\mathcal{S}_{pre}(o.f)$.

One of the cornerstones of the static semantics approach that we are presenting is how to express the invocation of a method. All the methods have the Stores as free variables. Since most of the statements can be considered as a method invocation with a special pre and post conditions, the semantics is defined only for that statement. Then, the other ones can be implemented as a method. An example of this concept is shown after Definition 5.

**Definition 5.** *The evaluation of a method call is defined as follows*[2]*:*

$$\text{S\_CALL\_1}\quad \frac{\begin{array}{l}\texttt{C.m} = \textbf{method}\ m(p_1 : T_1, ..., p_n : T_n)\ \textbf{returns}\ (ret : T_{ret})\\ \quad \textbf{requires}\ Pre;\ \textbf{modifies}\ Mods;\ \textbf{ensures}\ Pos;\ \{\ ...\ \}\\ typeof(e_0) == C\\ \models\ \forall \mathcal{S}_{pre} \bullet \mathbf{P}(\mathcal{S}_{pre}) \Rightarrow Pre[e_0/this, e_i/p_i](\mathcal{S}_{pre})\\ \models\ \forall \mathcal{S}_{pre}, \mathcal{S}_{pos} \bullet \exists id' \bullet (\mathbf{P}[id'/id](\mathcal{S}_{pre}) \wedge\\ \qquad Pos[e_0/this, e_i/p_i, id'/id, id/ret](\mathcal{S}_{pre}, \mathcal{S}_{pos}) \wedge alive(id, \mathcal{S}_{pos})) \Rightarrow\\ \qquad \mathbf{Q}(\mathcal{S}_{pre}, \mathcal{S}_{pos})\end{array}}{\{\mathbf{P}\}\ id := e_0.\texttt{m}(\texttt{e}_1, ..., \texttt{e}_n)\ \{\mathbf{Q}\}}$$

When invoking a method, we need to take a $\mathcal{S}_{pre}$ and $\mathcal{S}_{pos}$ that verifies the precondition and post-condition, respectively. We implicitly create a new variable $id$ that points to an element in the Store (the *result* reference of the rule) which is the return value of the method. The substitution operator $a/b$ should be interpreted as replacing $b$ with $a$.

For example, a natural division operation statement ($x := x/y$;) should be *statically* implemented as:

```
...
x := this.divide(x, y);
...
method divide(p1 : Num, p2 : Num) returns (ret : Num)
    requires p2 ≠ 0;
    ensures ret == p1/p2;
{
    ret := divideAux(p1, p2);
}
```

The *divideAux* method might implement the division as repeated subtractions. It is important to focus on the method's contract and the instantiation of the call rule (Definition 5):

**Precondition**

$$\begin{array}{l}\models\ \forall \mathcal{S}_{pre} \bullet \mathbf{P}(\mathcal{S}_{pre}) \Rightarrow Pre[\textbf{this}/this, x/p_1, y/p_2](\mathcal{S}_{pre})\\ \models\ \forall \mathcal{S}_{pre} \bullet \mathbf{P}(\mathcal{S}_{pre}) \Rightarrow (p_2 \neq 0)[\textbf{this}/this, x/p_1, y/p_2](\mathcal{S}_{pre})\\ \models\ \forall \mathcal{S}_{pre} \bullet \mathbf{P}(\mathcal{S}_{pre}) \Rightarrow (y \neq 0)(\mathcal{S}_{pre})\\ \models\ \forall \mathcal{S}_{pre} \bullet \mathbf{P}(\mathcal{S}_{pre}) \Rightarrow y \neq 0\end{array}$$

**Postcondition**

$$\models\ \forall \mathcal{S}_{pre}, \mathcal{S}_{pos} \bullet \exists id' \bullet (\mathbf{P}[id'/x](\mathcal{S}_{pre}) \wedge Pos[\textbf{this}\ this, x/p_1, y/p_2, id'/x, x/ret](\mathcal{S}_{pre}, \mathcal{S}_{pos}) \wedge alive(x, \mathcal{S}_{pos})) \Rightarrow \mathbf{Q}(\mathcal{S}_{pre}, \mathcal{S}_{pos})$$

[2]Since we are not support neither inheritance nor polymorphism, we know the type of $e_0$ before the execution

$$\models \forall \mathcal{S}_{pre}, \mathcal{S}_{pos} \bullet \exists id' \bullet (\mathbf{P}[id'/x](\mathcal{S}_{pre}) \wedge (ret == p_1/p_2)[\mathbf{this}/this, x/p_1, y/p_2, id'/x, x/ret](\mathcal{S}_{pre}, \mathcal{S}_{pos}) \wedge alive(x, \mathcal{S}_{pos})) \Rightarrow \mathbf{Q}(\mathcal{S}_{pre}, \mathcal{S}_{pos})$$

$$\models \forall \mathcal{S}_{pre}, \mathcal{S}_{pos} \bullet \exists id' \bullet (\mathbf{P}[id'/x](\mathcal{S}_{pre}) \wedge (x == id'/y)(\mathcal{S}_{pre}, \mathcal{S}_{pos}) \wedge alive(x, \mathcal{S}_{pos})) \Rightarrow \mathbf{Q}(\mathcal{S}_{pre}, \mathcal{S}_{pos})$$

$$\models \forall \mathcal{S}_{pre}, \mathcal{S}_{pos} \bullet \exists id' \bullet (\mathbf{P}[id'/x](\mathcal{S}_{pre}) \wedge (x == id'/y) \wedge alive(x, \mathcal{S}_{pos})) \Rightarrow \mathbf{Q}(\mathcal{S}_{pre}, \mathcal{S}_{pos})$$

For simplicity, this example shows the invocation of a method using only local variables. In order to include references to the store space in the specifications, we only need to use the free variables $\mathcal{S}_{pre}$ and $\mathcal{S}_{pos}$.

The first definition of the method invocation rule (S_CALL_1) is the one of a basic Java language. Like in the dynamic semantics, we need to add some conditions before and after the invocation takes place (i.e., checking the dynamic frame and verifying the footprint invariant). Therefore, the method invocation rule is extending like this:

**Definition 6.** *The evaluation of a method call is the following:*

$$\frac{\begin{array}{l} \texttt{C.m} = \mathbf{method}\ m(p_1 : T_1, ..., p_n : T_n)\ \mathbf{returns}\ (ret : T_{ret}) \\ \quad \mathbf{requires}\ Pre; \mathbf{modifies}\ Mod; \mathbf{ensures}\ Pos;\ \{\ ...\ \} \\ typeof(e_0) == C \\ \models\ \forall \mathcal{S}_{pre} \bullet \mathbf{P}(\mathcal{S}_{pre}) \Rightarrow (Pre[e_0/this, e_i/p_i](\mathcal{S}_{pre}) \wedge \\ \quad FootprintInvariant(\mathcal{S}_{pre}, e_0)) \\ \models\ \forall \mathcal{S}_{pre}, \mathcal{S}_{pos} \bullet \exists id' \bullet (\mathbf{P}[id'/id](\mathcal{S}_{pre}) \wedge \\ \quad HeapSucc(\mathcal{S}_{pre}, \mathcal{S}_{pos}, Mod[e_0/this, id'/id, e_i/p_i]) \wedge \\ \quad Pos[e_0/this, e_i/p_i, id'/id, id/ret](\mathcal{S}_{pre}, \mathcal{S}_{pos}) \wedge alive(id, \mathcal{S}_{pos}) \wedge \\ \quad FootprintInvariant(\mathcal{S}_{pos}, e_0) \wedge FootprintInvariant(\mathcal{S}_{pos}, e_i)) \\ \qquad \Rightarrow \mathbf{Q}(\mathcal{S}_{pre}, \mathcal{S}_{pos}) \end{array}}{\{\mathbf{P}\}\ id = e_0.\texttt{m}(\texttt{e}_1, ..., \texttt{e}_n)\ \{\mathbf{Q}\}}\ \text{S\_CALL\_2}$$

$$HeapSucc(\mathcal{S}_{pre}, \mathcal{S}_{pos}, Mod) \equiv (\forall x \bullet alive(x, \mathcal{S}_{pre}) \Rightarrow alive(x, \mathcal{S}_{pos})) \wedge (\forall x \bullet \mathcal{S}_{pre}(x) == \mathcal{S}_{pos}(x) \vee x \in Mod \vee \neg alive(x, \mathcal{S}_{pre}))$$

$$FootprintInvariant(\mathcal{S}, x) \equiv (isAcyclicQual(x) \Rightarrow footprint(\mathcal{S}, x, x) == \bar{1}) \wedge (\neg isAcyclicQual(x) \Rightarrow footprint(\mathcal{S}, x, x) \geq \bar{1}) \wedge (\forall f \bullet footprint(\mathcal{S}, x, \mathcal{S}(x.f)) \geq \bar{1} \wedge (\forall ref \bullet footprint(\mathcal{S}, x, ref) \geq footprint(\mathcal{S}, \mathcal{S}(x.f), ref)) \wedge FootprintInvariant(\mathcal{S}, \mathcal{S}(x.f))$$

When allocating a new instance, we need to make sure that the footprints of all the objects are correct (i.e., not reaching the actual new instance).

**Definition 7.** *The statement* $x :=$ **new** C(); *is expressed using the following method*

$$\begin{array}{l} \mathbf{method}\ allocate(C : ClassName)\ \mathbf{returns}\ (x : Ref) \\ \quad \mathbf{requires\ true}; \\ \quad \mathbf{modifies}\ \{\}; \\ \quad \mathbf{ensures}\ x \neq \mathbf{null} \wedge new(\mathcal{S}_{pre}, \mathcal{S}_{pos}, x); \\ \quad \mathbf{ensures}\ NewObjectFootprint(\mathcal{S}_{pre}, \mathcal{S}_{pos}, x); \end{array}$$

$$NewObjectFootprint(\mathcal{S}_{pre}, \mathcal{S}_{pos}, x) \equiv footprint(\mathcal{S}_{pos}, x, x) == \bar{1} \wedge (\forall ref_1 \bullet ref_1 \neq x \Rightarrow footprint(\mathcal{S}_{pos}, x, ref_1) == \bar{0} \wedge footprint(\mathcal{S}_{pos}, ref_1, x) == \bar{0}) \wedge (\forall ref_1, ref_2 \bullet ref_1 \neq x \wedge ref_2 \neq x \Rightarrow footprint(\mathcal{S}_{pre}, ref_1, ref_2) == footprint(\mathcal{S}_{pos}, ref_1, ref_2))$$

The method that reads an object field has a simple contract: it only needs to ensure that the result is the value of the desired field. Since it does not modify the memory, the modifies clause is empty.

**Definition 8.** *The statement* $x := o.field$; *is expressed using the following method*

$$\begin{array}{l}\textbf{method } read(o : Ref, field : Field) \textbf{ returns } (x : Ref)\\ \quad \textbf{requires } o \neq \textbf{null};\\ \quad \textbf{modifies } \{\};\\ \quad \textbf{ensures } x == \mathcal{S}_{pre}(o.field);\end{array}$$

When writing a field we need to ensure that the footprint values are correctly updated and that, when we are dealing with an acyclic object, the assignment does not generate a cycle. By now it should be clear that when an assignment is executed, there's an extensive impact on the footprint of (in worst case) all living objects.

**Definition 9.** *The statement* $o.field := x$; *is expressed using the following method:*

$$\begin{array}{l}\textbf{method } write(o : Ref, field : Field, x : Ref)\\ \quad \textbf{requires } o \neq \textbf{null} \wedge ((isAcyclicQual(o) \wedge x \neq \textbf{null}) \Rightarrow isAcyclicQual(x) \wedge\\ \qquad footprint(\mathcal{S}_{pre}, x, o) == \bar{0});\\ \quad \textbf{modifies } o.footprint;\\ \quad \textbf{ensures } x == \mathcal{S}_{pos}(o.field);\\ \quad \textbf{ensures } MergeFootprints(\mathcal{S}_{pre}, \mathcal{S}_{pos}, o, x);\end{array}$$

$$\begin{array}{l}MergeFootprints(\mathcal{S}_{pre}, \mathcal{S}_{pos}, o, x) \equiv (\forall ref_1, ref_2 \bullet (footprint(\mathcal{S}_{pre}, ref_1, o) == \bar{0} \Rightarrow\\ \quad footprint(\mathcal{S}_{pre}, ref_1, ref_2) == footprint(\mathcal{S}_{pos}, ref_1, ref_2)) \wedge\\ \quad ((footprint(\mathcal{S}_{pre}, ref_1, o) > \bar{0} \wedge \mathcal{S}_{pre}(o.field) \neq \textbf{null} \wedge x \neq \textbf{null}) \Rightarrow\\ \qquad footprint(\mathcal{S}_{pos}, ref_1, ref_2) == footprint(\mathcal{S}_{pre}, ref_1, ref_2) \dot{+} footprint(\mathcal{S}_{pre}, x, ref_2) \dot{-}\\ \qquad\quad footprint(\mathcal{S}_{pre}, \mathcal{S}_{pre}(o.field), ref_2)) \wedge\\ \quad ((footprint(\mathcal{S}_{pre}, ref_1, o) > \bar{0} \wedge \mathcal{S}_{pre}(o.field) \neq \textbf{null} \wedge x == \textbf{null}) \Rightarrow\\ \qquad footprint(\mathcal{S}_{pos}, ref_1, ref_2) == footprint(\mathcal{S}_{pre}, ref_1, ref_2) \dot{-}\\ \qquad\quad footprint(\mathcal{S}_{pre}, \mathcal{S}_{pre}(o.field), ref_2)) \wedge\\ \quad ((footprint(\mathcal{S}_{pre}, ref_1, o) > \bar{0} \wedge \mathcal{S}_{pre}(o.field) == \textbf{null} \wedge x \neq \textbf{null}) \Rightarrow\\ \qquad footprint(\mathcal{S}_{pos}, ref_1, ref_2) == footprint(\mathcal{S}_{pre}, ref_1, ref_2) \dot{+} footprint(\mathcal{S}_{pre}, x, ref_2)))\end{array}$$

This last rule maps with the operational semantics Rule D_FIELD_W_2 from Section 4.2.1. As we mentioned there, execution a field assignment statement would be a very expensive operation because, in worst case, it modifies all the living objects' footprint. Since the footprint is only used by the verifier, when building a compiler (and if the program verifies successfully) all the footprint operations and assertions can be omitted in order to get the desired performance in the execution.

**Definition 10.** *The conditional, sequence and empty statements execution is the following:*

$$\text{S\_SEQ}\quad \frac{\{\mathbf{P_1}\}\ stmt_1\ \{\mathbf{Q_1}\} \qquad \{\mathbf{P_2}\}\ stmt_2\ \{\mathbf{Q_2}\}}{\models \mathbf{P} \Rightarrow \mathbf{P_1} \qquad \models \mathbf{Q_1} \Rightarrow \mathbf{P_2} \qquad \models \mathbf{Q_2} \Rightarrow \mathbf{Q}}{\{\mathbf{P}\}\ stmt_1; stmt_2\ \{\mathbf{Q}\}} \qquad \text{S\_SKIP}\quad \frac{\models \mathbf{P} \Rightarrow \mathbf{Q}}{\{\mathbf{P}\}\ \textbf{skip}\ \{\mathbf{Q}\}}$$

$$\text{S\_IF}\quad \frac{\begin{array}{c}\{\mathbf{P_1}\}\ stmt_1\ \{\mathbf{Q_1}\} \qquad \{\mathbf{P_2}\}\ stmt_2\ \{\mathbf{Q_2}\}\\ \models \mathbf{P} \wedge expr \neq \textbf{null} \Rightarrow \mathbf{P_1} \qquad \models \mathbf{Q_1} \Rightarrow \mathbf{Q}\\ \models \mathbf{P} \wedge expr == \textbf{null} \Rightarrow \mathbf{P_2} \qquad \models \mathbf{Q_2} \Rightarrow \mathbf{Q}\end{array}}{\{\mathbf{P}\}\ \textbf{if}\ (expr)\ \{stmt_1\}\ \textbf{else}\ \{stmt_2\}\ \{\mathbf{Q}\}}$$

## 4.5 Semantics Soundness

The previous sections focused on the definition of the dynamic and static semantics for the language that we are introducing. This section intends to prove some particular invariants and properties using those definitions. By the end of the demonstrations, we expect to show how all the decisions that we have made for our language will converge in the verification of the acyclic data structures.

**Definition 11.** *A store* $\sigma$ *is defined as* ***valid*** *if:*

- *For all Refs* $ref_1$ *and* $ref_2$, $footprint(\sigma, ref_1, ref_2) == n$ *implies that there are* $n$ *different paths from* $ref_1$ *to* $ref_2$ *in* $\sigma$
- **null** *is not in the domain of* $\sigma$

**Theorem 1.** *For all valid store* $\sigma$, *environment* $env$, *reference* $this$, *and blocks* $\{\mathbf{P}\}$ $Stmt$ $\{\mathbf{Q}\}$, *if* $\mathsf{fails}(\sigma, env, this, Stmt)$, *then* $\sigma, env, this \not\models \mathbf{P}$.

*Proof.* We are going to prove the theorem using induction over $Stmt$. Most of the cases are going to use the definitions from Sections 4.2, 4.3 and 4.4. Since most of the statements from the base cases are executed as a method call, remember the precondition premise (rule 5):

$\forall \mathcal{S}_{pre} \bullet \mathbf{P}(\mathcal{S}_{pre}) \Rightarrow Pre[y/\mathbf{this}, val_i/p_i](\mathcal{S}_{pre})$

That's why if we prove that the method precondition is not satisfied (knowing that the fails predicate holds), then we prove that $\sigma, env, this \not\models \mathbf{P}$.

The base cases are:

- $\{\mathbf{P}\}$ $x :=$ **new** C() $\{\mathbf{Q}\}$
  Since the fails predicate does not hold for the allocation statement, the theorem is trivially valid.
- $\{\mathbf{P}\}$ $o.field := x$ $\{\mathbf{Q}\}$
  First let's recall the premise of the fails rule for the field write (rule 4.10):

  $\kappa \neq \mathbf{null}$ and $isAcyclicQual(\iota)$ and $\sigma(\kappa, \texttt{Footprint})[\iota] > 0$

  In order to verify if the operation precondition is not satisfied, we need to interpret (rule 9):

  $$v_s, v_r, this \models o \neq \mathbf{null} \wedge ((isAcyclicQual(o) \wedge x \neq \mathbf{null}) \Rightarrow isAcyclicQual(x) \wedge footprint(\mathcal{S}_{pre}, x, o) == \bar{0})$$

  Where $v_r$ maps the variable names to the references in $\sigma$, and $v_{\mathcal{S}} = \{\mathcal{S}_{pre} \mapsto \sigma\}$

  Taking each part of the formula:

  – $v_s, v_r, this \models o \neq \mathbf{null} \Leftrightarrow \mathcal{I}^{Ref}_{v_s,v_r,this}(o) \neq \mathcal{I}^{Ref}_{v_s,v_r,this}(\mathbf{null}) \Leftrightarrow o$ is not null

  – $v_s, v_r, this \models ((isAcyclicQual(o) \wedge x \neq \mathbf{null}) \Rightarrow isAcyclicQual(x) \wedge footprint(\mathcal{S}_{pre}, x, o) == \bar{0}) \Leftrightarrow$
  $((v_s, v_r, this \models isAcyclicQual(o)) \wedge \mathcal{I}^{Ref}_{v_s,v_r,this}(x) \neq \mathcal{I}^{Ref}_{v_s,v_r,this}(\mathbf{null}))$ implies
  $\mathcal{I}^{Num}_{v_s,v_r,this}(footprint(\mathcal{S}_{pre}, x, o)) = \mathcal{I}^{Num}_{v_s,v_r,this}(\bar{0}) \Leftrightarrow$
  ($o$ type is annotated as acyclic and $x \neq \mathbf{null}$) implies $x$ type is annotated as acyclic and $\sigma(x, \texttt{Footprint})[o] = 0$

  Since the fails predicate states that if $o$ type is annotated as acyclic and $x \neq \mathbf{null}$, then $\sigma(x, \texttt{Footprint})[o] > 0$; and the operation precondition requires that $\sigma(x, \texttt{Footprint})[o] = 0$, in consequence, $\sigma, env, this \not\models \mathbf{P}$.
- $\{\mathbf{P}\}$ $x := o.field$ $\{\mathbf{Q}\}$
  When reading an object field, the fails predicate holds for (Figure 4.8):

  $o$ is **null**

  The static precondition of a field read is **requires** $o \neq \mathbf{null}$ (Definition 8). Therefore $\sigma, env, this \not\models \mathbf{P}$.
- $\{\mathbf{P}\}$ $id := e_0.\texttt{m}(e_1, ..., e_n)$ $\{\mathbf{Q}\}$
  From the fact that the fails predicate holds, we know that (rules in Figure 4.9):

  $\texttt{C.m} =$**method** $m(p_1 : T_1, ..., p_n : T_n)$ **returns** $(ret : T_{ret})$

**requires** $Pre$; **ensures** $Post$;
{ $Body$ }

$e_i, \sigma, env, this \overset{e}{\rightsquigarrow} \iota_i$

$env' = [p_1 \mapsto \iota_1, ..., p_n \mapsto \iota_n, ret \mapsto \textbf{null}]$

$\sigma, env, this \not\models Pre$

$\sigma, env \not\models FootprintInvariant$ or $\sigma', env' \not\models FootprintInvariant$

Following the static invocation rule [6], first we need to analyze the precondition:

$$v_s, v_r, this \models \forall \mathcal{S}_{pre} \bullet \mathbf{P}(\mathcal{S}_{pre}) \Rightarrow Pre[e_0/\textbf{this}, e_i/p_i](\mathcal{S}_{pre}) \wedge FootprintInvariant(\mathcal{S}_{pre}, e_0) \wedge FootprintInvariant(\mathcal{S}_{pre}, e_i)$$

Where $v_r$ maps the variables defined in $env'$ and $v_s = \{\mathcal{S}_{pre} \mapsto \sigma\}$.

$$v_s, v_r, this \models \forall \mathcal{S}_{pre} \bullet \mathbf{P}(\mathcal{S}_{pre}) \Rightarrow Pre[e_0/\textbf{this}, e_i/p_i](\mathcal{S}_{pre}) \wedge FootprintInvariant(\mathcal{S}_{pre}, e_0) \wedge FootprintInvariant(\mathcal{S}_{pre}, e_i) \Leftrightarrow$$
$$v_s[\mathcal{S}_{pre} \mapsto h], v_r, this \models \mathbf{P}(\mathcal{S}_{pre}) \Rightarrow Pre[e_0/\textbf{this}, e_i/p_i](\mathcal{S}_{pre}) \wedge FootprintInvariant(\mathcal{S}_{pre}, e_0) \wedge FootprintInvariant(\mathcal{S}_{pre}, e_i) \text{ for all } h \in Store \Leftrightarrow$$
$$v_s[\mathcal{S}_{pre} \mapsto h], v_r, this \models \mathbf{P}(\mathcal{S}_{pre}) \text{ implies that } v_s[\mathcal{S}_{pre} \mapsto h], v_r \models Pre[e_0/\textbf{this}, e_i/p_i] \text{ and } v_s[\mathcal{S}_{pre} \mapsto h], v_r, this \models FootprintInvariant(\mathcal{S}_{pre}, e_0) \wedge FootprintInvariant(\mathcal{S}_{pre}, e_i) \text{ for all } h \in Store \Leftrightarrow$$
$$\sigma, env', \iota_0 \models Pre \text{ and } \sigma, env \models FootprintInvariant.$$

Contradicting the premise of the fails predicate. Therefore $\sigma, env, this \not\models \mathbf{P}$.

- $\{\mathbf{P}\}$ **skip**$\{\mathbf{Q}\}$
  If $Stmt$ is **skip** (the empty sentence), the theorem is valid trivially.

The inductive cases are:

- $\{\mathbf{P}\}$ $stmt_1; stmt_2$ $\{\mathbf{Q}\}$
  Recalling fails predicate premises for the sequencing of statements (rules 4.8):

  $\mathsf{fails}(\sigma, env, this, stmt_1)$, or

  $stmt_1, \sigma, env, this \rightsquigarrow \sigma', env'$ and $\mathsf{fails}(\sigma', env', this, stmt_2)$

  The static semantics sequencing rule (Definition 10) states that:

$$\frac{\{\mathbf{P_1}\}\ stmt_1\ \{\mathbf{Q_1}\} \quad \{\mathbf{P_2}\}\ stmt_2\ \{\mathbf{Q_2}\}}{\models \mathbf{P} \Rightarrow \mathbf{P_1} \quad \models \mathbf{Q_1} \Rightarrow \mathbf{P_2} \quad \models \mathbf{Q_2} \Rightarrow \mathbf{Q}}$$

  Therefore, using the inductive hypothesis, we know that:

  If $\mathsf{fails}(\sigma, env, this, stmt_1)$, then $\sigma, env, this \not\models \mathbf{P_1}$, implying that $\mathbf{P}$ is false

  If $\mathsf{fails}(\sigma', env', this, stmt_2)$, then $\sigma', env', this \not\models \mathbf{P_2}$, implying that $\mathbf{Q_1}$, $\mathbf{P_1}$ and $\mathbf{P}$ are false

  Then $\sigma, env, this \not\models \mathbf{P}$.

- $\{\mathbf{P}\}$ **if** $(cond)$ $\{stmt_1\}$ **else** $\{stmt_2\}$ $\{\mathbf{Q}\}$
  Let's break each case of the conditional.

  The fails predicate premise defined for the true conditional case is (rules 4.8):

  $cond, \sigma, env, this \overset{e}{\rightsquigarrow} \iota$

  $\iota \neq \textbf{null}$ and $\mathsf{fails}(\sigma, env, this, stmt_1)$

The static semantics rule for the conditional is (Definition 10):

$$\begin{array}{cc} \{\mathbf{P_1}\}\ stmt_1\ \{\mathbf{Q_1}\} & \{\mathbf{P_2}\}\ stmt_2\ \{\mathbf{Q_2}\} \\ \models \mathbf{P} \wedge expr \neq \mathbf{null} \Rightarrow \mathbf{P_1} & \models \mathbf{Q_1} \Rightarrow \mathbf{Q} \\ \models \mathbf{P} \wedge expr == \mathbf{null} \Rightarrow \mathbf{P_2} & \models \mathbf{Q_2} \Rightarrow \mathbf{Q} \end{array}$$

Using the inductive hypothesis we know that:

$\sigma, env, this \models (\mathbf{P} \wedge cond \neq \mathbf{null} \Rightarrow \mathbf{P_1})$, contradicting the premise of the fails predicate

Then the false case is proven the same way:

The fails predicate premise defined for the false conditional case is (rules 4.8):

$cond, \sigma, env, this \overset{e}{\rightsquigarrow} \iota$

$\iota = \mathbf{null}$ and $\mathsf{fails}(\sigma, env, this, stmt_1)$

Using the inductive hypothesis and rule 10, we know that:

$\sigma, env, this \models (\mathbf{P} \wedge cond == \mathbf{null} \Rightarrow \mathbf{P_2})$, contradicting the premise of the fails predicate

Then $\sigma, env, this \not\models \mathbf{P}$.

□

**Corollary 1.** *Theorem 1 is equivalent to prove that for all valid store $\sigma$, environment env, reference this, and blocks $\{\mathbf{P}\}$ Stmt $\{\mathbf{Q}\}$, if $\sigma, env, this \models \mathbf{P}$, then $\mathsf{fails}(\sigma, env, this, Stmt)$ is not defined.*

**Theorem 2.** *For all valid Store $\sigma$, variables env and blocks $\{\mathbf{P}\}$ Stmt $\{\mathbf{Q}\}$, if:*

- $\sigma, env, this \models \mathbf{P}$, *and*
- $Stmt, \sigma, env, this \rightsquigarrow \sigma', env'$

*then $\{\sigma, \sigma'\}, env', this \models \mathbf{Q}$ and $\sigma'$ is valid.*

*Proof.* We are going to prove the theorem using induction over *Stmt*. Again, since most of the statements from the bases cases are executed as a method call, remember the postcondition premise (simplified rule 5):

$\models\ \forall \mathcal{S}_{pre}, \mathcal{S}_{pos} \bullet (\mathbf{P}(\mathcal{S}_{pre}) \wedge Pos[y/\mathbf{this}, val_i/p_i, id/ret](\mathcal{S}_{pre}, \mathcal{S}_{pos})) \Rightarrow \mathbf{Q}(\mathcal{S}_{pre}, \mathcal{S}_{pos})$

then in those cases we only need to prove that the method postcondition is satisfied.

The base cases are.

- $\{\mathbf{P}\}\ x := \mathbf{new}\ \mathtt{C}()\ \{\mathbf{Q}\}$

  In this case we need to ensure that the dynamic post-state implies the static postcondition. The effect on the resulting state ($\sigma'$) is the following (rule in Section 4.2.1):

  – $\sigma' = \sigma[\iota \mapsto\ll \mathtt{f_1} : \mathbf{null}, ..., \mathtt{f_n} : \mathbf{null}, \mathtt{Footprint} : \{\iota : 1\} \gg^{\mathtt{C}}]$

  – $env' = env[x \mapsto \iota]$

  Then we need to verify that (rule 7):

$$\begin{array}{l} v_s, v_r, this \models x \neq \mathbf{null} \wedge \neg alive(x, \mathcal{S}_{pre}) \wedge alive(x, \mathcal{S}_{pos}) \wedge footprint(\mathcal{S}_{pos}, x, x) == \bar{1} \wedge \\ \quad (\forall ref_1 \bullet ref_1 \neq x \Rightarrow (footprint(\mathcal{S}_{pos}, x, ref_1) == \bar{0} \wedge \\ \quad\quad footprint(\mathcal{S}_{pos}, ref_1, x) == \bar{0})) \end{array}$$

  Where $v_r$ maps the variable names to the references in $\sigma$ and $\sigma'$ (defined in $env'$), and $v_{\mathcal{S}} = \{\mathcal{S}_{pre} \mapsto \sigma,\ \mathcal{S}_{pos} \mapsto \sigma'\}$

  Let's analyze the expression by parts:

- $v_s, v_r, this \models \neg alive(x, \mathcal{S}_{pre}) \Leftrightarrow \mathcal{I}^{Ref}_{v_s,v_r,this}(x)$ is not **null** and is not in the domain of $\mathcal{I}^{Store}_{v_s,v_r,this}(\mathcal{S}_{pre}) \Leftrightarrow$ $\iota$ is not  and $\iota$ is not in the domain to $\sigma$, which is guaranteed because $\iota$ is fresh
- $v_s, v_r \models alive(x, \mathcal{S}_{pos}) \Leftrightarrow \mathcal{I}^{Ref}_{v_s,v_r,this}(x)$ is **null** or is in the domain of $\mathcal{I}^{Store}_{v_s,v_r,this}(\mathcal{S}_{pos}) \Leftrightarrow \iota$ is not  implies that $\iota$ is in the domain to $\sigma'$, which is guaranteed because $\sigma' =$ $\sigma[\iota \mapsto\ll \mathtt{f_1} : \textbf{null}, ..., \mathtt{f_n} : \textbf{null} \gg^{\mathtt{C}}]$
- $v_s, v_r, this \models footprint(\mathcal{S}_{pos}, x, x) == \bar{1} \Leftrightarrow \mathcal{I}^{Num}_{v_s,v_r,this}(footprint(\mathcal{S}_{pos}, x, x)) =$ $\mathcal{I}^{Num}_{v_s,v_r,this}(\bar{1}) \Leftrightarrow \sigma'(\iota, \mathtt{Footprint})[\iota] = 1$
- $v_s, v_r, this \models (\forall ref_1 \bullet ref_1 \neq x \Rightarrow (footprint(\mathcal{S}_{pos}, x, ref_1) == \bar{0}) \wedge$ $footprint(\mathcal{S}_{pos}, ref_1, x) == \bar{0}) \Leftrightarrow$ $v_s, v_r[ref_1 \mapsto \iota_1], this \models ref_1 \neq x \Rightarrow (footprint(\mathcal{S}_{pos}, x, ref_1) == \bar{0} \wedge$ $footprint(\mathcal{S}_{pos}, ref_1, x) == \bar{0})$ for all $\iota_1 \in Ref \Leftrightarrow$ $\mathcal{I}^{Ref}_{v_s,v_r[ref_1\mapsto\iota_1],this}(ref_1) \neq \mathcal{I}^{Ref}_{v_s,v_r[ref_1\mapsto\iota_1],this}(x)$ implies that $\mathcal{I}^{Num}_{v_s,v_r[ref_1\mapsto\iota_1],this}(footprint(\mathcal{S}_{pos}, x, ref_1)) = \mathcal{I}^{Num}_{v_s,v_r[ref_1\mapsto\iota_1],this}(\bar{0})$ and $\mathcal{I}^{Num}_{v_s,v_r[ref_1\mapsto\iota_1],this}(footprint(\mathcal{S}_{pos}, x, ref_1)) = \mathcal{I}^{Num}_{v_s,v_r[ref_1\mapsto\iota_1],this}(\bar{0})$ for all $\iota_1 \in Ref \Leftrightarrow$ for all $\iota_1, \iota_1 \neq \iota$ implies that $\sigma'(\iota, \mathtt{Footprint})[\iota_1] = 0$ and $\sigma'(\iota_1, \mathtt{Footprint})[\iota] = 0$

  This last condition is guaranteed because the dynamic semantics does not modify any reference other than $\iota$ and $\iota$'s multiset is initialized with 0's.

- Since the new object is only accessible from itself and the key added to $\sigma$ is newly created (not **null**), this predicate makes the Store valid from $x$'s perspective
- Using the inductive hypothesis and considering that the rest of the elements of the Store and footprint are untouched, the Store is valid for the remaining objects

Then we conclude that an allocation execution verifies its postcondition and the resulting Store is valid.

- $\{\mathbf{P}\}\ o.field := x\ \{\mathbf{Q}\}$

For simplicity, let's assume that x is not null. The effect on the state after the execution is the following (rule 4.7):

$\sigma' = \sigma[o, field \mapsto x]$

$\sigma(o, field) \neq \textbf{null}$ implies that for all $\iota_1, \iota_2,\ \sigma(\iota_1, \mathtt{Footprint})[o] > 0$ implies that $\sigma'(\iota_1, \mathtt{Footprint})[\iota_2] = \sigma(\iota_1, \mathtt{Footprint})[\iota_2] - \sigma(o, field)(\mathtt{Footprint})[\iota_2]$ $+\sigma(x, \mathtt{Footprint})[\iota_2]$

$\sigma(o, field) = \textbf{null}$ implies that for all $\iota_1, \iota_2,\ \sigma(\iota_1, \mathtt{Footprint})[o] > \bar{0}$ implies that $\sigma'(\iota_1, \mathtt{Footprint})[\iota_2] = \sigma(\iota_1, \mathtt{Footprint})[\iota_2] + \sigma(x, \mathtt{Footprint})[\iota_2]$

Then we need to verify that (rule 9):

$v_s, v_r, this \models x == \mathcal{S}_{pos}(o.field) \wedge \forall ref_1, ref_2 \bullet ((footprint(\mathcal{S}_{pre}, ref_1, o) > \bar{0} \wedge$ $\mathcal{S}_{pre}(o.field) \neq \textbf{null}) \Rightarrow footprint(\mathcal{S}_{pos}, ref_1, ref_2) == footprint(\mathcal{S}_{pre}, ref_1, ref_2)$ $\dot{+} footprint(\mathcal{S}_{pre}, x, ref_2) \dot{-} footprint(\mathcal{S}_{pre}, \mathcal{S}_{pre}(o.field), ref_2)) \wedge$ $((footprint(\mathcal{S}_{pre}, ref_1, o) > \bar{0} \wedge \mathcal{S}_{pre}(o.field) == \textbf{null}) \Rightarrow footprint(\mathcal{S}_{pos}, ref_1, ref_2)$ $== footprint(\mathcal{S}_{pre}, ref_1, ref_2) \dot{+} footprint(\mathcal{S}_{pre}, x, ref_2))$

Where $v_r$ maps the variable names to the references in $\sigma$ and $\sigma'$, and $v_{\mathcal{S}} = \{\mathcal{S}_{pre} \mapsto \sigma,\ \mathcal{S}_{pos} \mapsto \sigma'\}$

Breaking each part of the formula and taking into account that all the references in $\sigma'$ that are not changed in the rule have the value from $\sigma$:

- $v_s, v_r, this \models x == \mathcal{S}_{pos}(o.field) \Leftrightarrow \mathcal{I}^{Ref}_{v_s,v_r}(x) = \mathcal{I}^{Ref}_{v_s,v_r,this}(\mathcal{S}_{pos}(o.field)) \Leftrightarrow$ $x = \mathcal{I}^{Store}_{v_s,v_r,this}(\mathcal{S}_{pos})(\mathcal{I}^{Ref}_{v_s,v_r,this}(o), \mathcal{I}^{Field}_{v_s,v_r,this}(field)) \Leftrightarrow x = \sigma'(o, field) \Leftrightarrow$ $x = \sigma[o, field \mapsto x](o, field) \Leftrightarrow x = x$

– $v_s, v_r, this \models \forall ref_1, ref_2 \bullet ((footprint(\mathcal{S}_{pre}, ref_1, o) > \bar{0} \wedge \mathcal{S}_{pre}(o.field) \neq \textbf{null}) \Rightarrow$
$footprint(\mathcal{S}_{pos}, ref_1, ref_2) == footprint(\mathcal{S}_{pre}, ref_1, ref_2) \dot{+}$
$footprint(\mathcal{S}_{pre}, x, ref_2) \dot{-} footprint(\mathcal{S}_{pre}, \mathcal{S}_{pre}(o.field), ref_2)) \Leftrightarrow$
$v_s, v_r[ref_1 \mapsto \iota_1, ref_2 \mapsto \iota_2], this \models ((footprint(\mathcal{S}_{pre}, ref_1, o) > \bar{0} \wedge$
$\mathcal{S}_{pre}(o.field) \neq \textbf{null}) \Rightarrow footprint(\mathcal{S}_{pos}, ref_1, ref_2) ==$
$footprint(\mathcal{S}_{pre}, ref_1, ref_2) \dot{+} footprint(\mathcal{S}_{pre}, x, ref_2) \dot{-}$
$footprint(\mathcal{S}_{pre}, \mathcal{S}_{pre}(o.field), ref_2))$ for all $\iota_1, \iota_2 \in Ref \Leftrightarrow$
$((\mathcal{I}^{Num}_{v_s, v_r[ref_1 \mapsto \iota_1, ref_2 \mapsto \iota_2], this}(footprint(\mathcal{S}_{pre}, ref_1, o)) > \mathcal{I}^{Num}_{v_s, v_r[ref_1 \mapsto \iota_1, ref_2 \mapsto \iota_2], this}(\bar{0}) \wedge$
$\mathcal{I}^{Ref}_{v_s, v_r[ref_1 \mapsto \iota_1, ref_2 \mapsto \iota_2], this}(\mathcal{S}_{pre}(o.field)) \neq \mathcal{I}^{Ref}_{v_s, v_r[ref_1 \mapsto \iota_1, ref_2 \mapsto \iota_2], this}(\textbf{null})) \Rightarrow$
$\mathcal{I}^{Num}_{v_s, v_r[ref_1 \mapsto \iota_1, ref_2 \mapsto \iota_2], this}(footprint(\mathcal{S}_{pos}, ref_1, ref_2)) =$
$\mathcal{I}^{Num}_{v_s, v_r[ref_1 \mapsto \iota_1, ref_2 \mapsto \iota_2], this}(footprint(\mathcal{S}_{pre}, ref_1, ref_2)) +$
$\mathcal{I}^{Num}_{v_s, v_r[ref_1 \mapsto \iota_1, ref_2 \mapsto \iota_2], this}(footprint(\mathcal{S}_{pre}, x, ref_2)) -$
$\mathcal{I}^{Num}_{v_s, v_r[ref_1 \mapsto \iota_1, ref_2 \mapsto \iota_2], this}(footprint(\mathcal{S}_{pre}, \mathcal{I}^{Ref}_{v_s, v_r[ref_1 \mapsto \iota_1, ref_2 \mapsto \iota_2], this}($
$\mathcal{S}_{pre}(o.field)), ref_2)))$ for all $\iota_1, \iota_2 \in Ref \Leftrightarrow$
for all $\iota_1, \iota_2,\ \sigma'(\iota_1, \texttt{Footprint})[o] > 0$ and $\sigma'(o, field) \neq \textbf{null}$ implies
$\sigma'(\iota_1, \texttt{Footprint})[\iota_2] = \sigma(\iota_1, \texttt{Footprint})[\iota_2] - \sigma(o, field)(\texttt{Footprint})[\iota_2] +$
$\sigma(x, \texttt{Footprint})[\iota_2]$

– $v_s, v_r, this \models \forall ref_1, ref_2 \bullet ((footprint(\mathcal{S}_{pre}, ref_1, o) > \bar{0} \wedge \mathcal{S}_{pre}(o.field) = \textbf{null}) \Rightarrow$
$footprint(\mathcal{S}_{pos}, ref_1, ref_2) == footprint(\mathcal{S}_{pre}, ref_1, ref_2) \dot{+}$
$footprint(\mathcal{S}_{pre}, x, ref_2)) \Leftrightarrow$
$v_s, v_r[ref_1 \mapsto \iota_1, ref_2 \mapsto \iota_2], this \models ((footprint(\mathcal{S}_{pre}, ref_1, o) > \bar{0} \wedge$
$\mathcal{S}_{pre}(o.field) == \textbf{null}) \Rightarrow footprint(\mathcal{S}_{pos}, ref_1, ref_2) ==$
$footprint(\mathcal{S}_{pre}, ref_1, ref_2) \dot{+} footprint(\mathcal{S}_{pre}, x, ref_2))$ for all $\iota_1, \iota_2 \in Ref \Leftrightarrow$
$((\mathcal{I}^{Num}_{v_s, v_r[ref_1 \mapsto \iota_1, ref_2 \mapsto \iota_2], this}(footprint(\mathcal{S}_{pre}, ref_1, o)) > \mathcal{I}^{Num}_{v_s, v_r[ref_1 \mapsto \iota_1, ref_2 \mapsto \iota_2], this}(\bar{0}) \wedge$
$\mathcal{I}^{Ref}_{v_s, v_r[ref_1 \mapsto \iota_1, ref_2 \mapsto \iota_2], this}(\mathcal{S}_{pre}(o.field)) = \mathcal{I}^{Ref}_{v_s, v_r[ref_1 \mapsto \iota_1, ref_2 \mapsto \iota_2], this}(\textbf{null})) \Rightarrow$
$\mathcal{I}^{Num}_{v_s, v_r[ref_1 \mapsto \iota_1, ref_2 \mapsto \iota_2], this}(footprint(\mathcal{S}_{pos}, ref_1, ref_2)) =$
$\mathcal{I}^{Num}_{v_s, v_r[ref_1 \mapsto \iota_1, ref_2 \mapsto \iota_2], this}(footprint(\mathcal{S}_{pre}, ref_1, ref_2)) +$
$\mathcal{I}^{Num}_{v_s, v_r[ref_1 \mapsto \iota_1, ref_2 \mapsto \iota_2], this}(footprint(\mathcal{S}_{pre}, x, ref_2)))$
for all $\iota_1, \iota_2 \in Ref \Leftrightarrow$
for all $\iota_1, \iota_2,\ \sigma'(\iota_1, \texttt{Footprint})[o] > 0$ and $\sigma'(o, field) = \textbf{null}$ implies
$\sigma'(\iota_1, \texttt{Footprint})[\iota_2] = \sigma(\iota_1, \texttt{Footprint})[\iota_2]$

- The field write operation has a high impact on the validity of the Store. We are ensuring that the Store remains valid by:
  - Removing the elements of the previously stored field, since those references will not be reached anymore
  - Adding the new paths of the assigned reference
  - Not adding a key to $\sigma'$
- Since the precondition states that $o \neq \textbf{null}$, we are ensuring that **null** is not defined in $\mathcal{S}_{pos}$
- Using the inductive hypothesis and considering that the rest of the elements of the Store and footprint are untouched, the Store is valid for the remaining objects

Then we conclude that a field write execution verifies its postcondition and the resulting Store is valid.

- $\{\mathbf{P}\}\ x := o.field\ \{\mathbf{Q}\}$

  The effect on the state after the execution is (rules in Figure 4.6):

  $$env' = env[x \mapsto o.field]$$

  Then we need to verify that (rule 8):

  – $v_s, v_r, this \models x == \mathcal{S}_{pre}(o.field)$

Where $v_r$ maps the variable names to the references in $\sigma$ and $\sigma'$, and $v_{\mathcal{S}} = \{\mathcal{S}_{pre} \mapsto \sigma,\ \mathcal{S}_{pos} \mapsto \sigma'\}$

- $v_s, v_r, this \models x == \mathcal{S}_{pre}(o.field) \Leftrightarrow \mathcal{I}^{Ref}_{v_s,v_r,this}(x) = \mathcal{I}^{Ref}_{v_s,v_r,this}(\mathcal{S}_{pre}(o.field)) \Leftrightarrow$
  $\sigma(o.field) = \sigma(o.field)$

The operation has no impact over the footprint nor $\sigma$, using the inductive hypothesis, the Store remains valid.

Then we conclude that a field read execution verifies its postcondition and the resulting Store is valid.

- $\{\mathbf{P}\}\ id := e_0.\mathtt{m}(e_1, ..., e_n)\ \{\mathbf{Q}\}$

Recalling the dynamic semantics invocation rule and, using Corollary 1, that the fails predicate does not hold (rules in Figures 4.8 and 4.9), we know that:

$\mathtt{C.m}$ =**method** $m(p_1 : T_1, ..., p_n : T_n)$ **returns** $(ret : T_{ret})$
**requires** $Pre$; **modifies** $Mod$; **ensures** $Post$;
{ $Body$ }

$e_i, \sigma, env, this \overset{e}{\rightsquigarrow} \iota_i$

For all $e_{mod,i}$ in $Mods$, $e_{mod,i}, \sigma, env', \iota_0 \overset{e}{\rightsquigarrow} \iota_{mod,i}$

$CheckFrame(\sigma, \sigma', \iota_{mod,0}, ..., \iota_{mod,n})$

$\sigma, env \models FootprintInvariant$ and $\sigma', env' \models FootprintInvariant$

$env' = [p_1 \mapsto \iota_1, ..., p_n \mapsto \iota_n]$

$\sigma, env', \iota_0 \models Pre$

$Body', \sigma, env', \iota_0 \rightsquigarrow \sigma', env''$

$\sigma', env'', \iota_0 \models Post$

Using the definition of the static invocation rule [6], we need to verify that:
$v_s, v_r, this \models \forall \mathcal{S}_{pre}, \mathcal{S}_{pos} \bullet \exists id' \bullet (\mathbf{P}[id'/id](\mathcal{S}_{pre}) \wedge$
$HeapSucc(\mathcal{S}_{pre}, \mathcal{S}_{pos}, Mod[e_0/\mathbf{this}, id'/id, e_i/p_i]) \wedge$
$Pos[e_0/\mathbf{this}, e_i/p_i, id'/id, id/ret](\mathcal{S}_{pre}, \mathcal{S}_{pos}) \wedge alive(id, \mathcal{S}_{pos}) \wedge$
$FootprintInvariant(\mathcal{S}_{pos}, e_0) \wedge FootprintInvariant(\mathcal{S}_{pos}, e_i)) \Rightarrow \mathbf{Q}(\mathcal{S}_{pre}, \mathcal{S}_{pos}) \Leftrightarrow$ $v_s[\mathcal{S}_{pre} \mapsto h_{pre}, \mathcal{S}_{pos} \mapsto h_{pos}], v_r, this \models \exists id' \bullet (\mathbf{P}[id'/id](\mathcal{S}_{pre}) \wedge$
$HeapSucc(\mathcal{S}_{pre}, \mathcal{S}_{pos}, Mod[e_0/\mathbf{this}, id'/id, e_i/p_i]) \wedge$
$Pos[e_0/\mathbf{this}, e_i/p_i, id'/id, id/ret](\mathcal{S}_{pre}, \mathcal{S}_{pos}) \wedge alive(id, \mathcal{S}_{pos}) \wedge$
$FootprintInvariant(\mathcal{S}_{pos}, e_0) \wedge FootprintInvariant(\mathcal{S}_{pos}, e_i)) \Rightarrow \mathbf{Q}(\mathcal{S}_{pre}, \mathcal{S}_{pos})$
for all $h_{pre}, h_{pos} \in Store \Leftrightarrow$
$\sigma', env'', \iota_0 \models Post$ , and $CheckFrame(\sigma, \sigma', Mods)$, and $\sigma', env' \models FootprintInvariant$. In this case, $\sigma$ is the $Store$ that verifies the precondition
and $\sigma'$ the postcondition.

The validity of the resulting Store is guaranteed by:

- Ensuring that the footprint reflects that a reference reaches itself (from $FootprintInvariant$ macro):
  ($isAcyclicQual(\iota)$ implies that $\sigma(\iota, \mathtt{Footprint})[\iota] = 1$) and
  ($\neg isAcyclicQual(\iota)$ implies that $\sigma(\iota, \mathtt{Footprint})[\iota] \geq 1$)
- Ensuring that the footprint of a reference includes the ones of its fields (from $FootprintInvariant$ macro):
  (for all $f$ field in $\iota$, $\sigma(\iota, \mathtt{Footprint})[\sigma(\iota, f)] \geq 1$ and
  (for all $\iota_1$ $Ref$, $\sigma(\iota, \mathtt{Footprint})[\iota_1] \geq \sigma(\sigma(\iota, f), \mathtt{Footprint})[\iota_1]$ and
  $FootprintInvariant(\sigma, \sigma(\iota, f))$))
- $\sigma'$ is the successor of $\sigma$
- The result of the body is in the domain of $\sigma'$ if it's not null ($v_s, v_r, \mathtt{val_0} \models alive(id, \mathcal{S}_{pos})$)

Then we conclude that a method invocation execution verifies its postcondition and the resulting Store is valid.

- $\{\mathbf{P}\}$ **skip**$\{\mathbf{Q}\}$
  If *Stmt* is **skip** (the empty sentence), the theorem is valid trivially.

The inductive cases are:

- $\{\mathbf{P}\}$ $stmt_1; stmt_2$ $\{\mathbf{Q}\}$
  Using the inductive hypothesis, we know that $\{\mathbf{P_1}\}$ $stmt_1$; $\{\mathbf{Q_1}\}$ and $\{\mathbf{P_2}\}$ $stmt_2$; $\{\mathbf{Q_2}\}$ verify their postconditions and result in a valid Store. Using the rule from definition 10:

$$\begin{array}{ccc} \{\mathbf{P_1}\}\ stmt_1\ \{\mathbf{Q_1}\} & & \{\mathbf{P_2}\}\ stmt_2\ \{\mathbf{Q_2}\} \\ \models \mathbf{P} \Rightarrow \mathbf{P_1} & \models \mathbf{Q_1} \Rightarrow \mathbf{P_2} & \models \mathbf{Q_2} \Rightarrow \mathbf{Q} \end{array}$$

  We also know that $\mathbf{Q_1} \Rightarrow \mathbf{P_2}$ and $\mathbf{Q_2} \Rightarrow \mathbf{Q}$, so the resulting store $\sigma$' (that verifies $\mathbf{Q}$) will also be valid.

- $\{\mathbf{P}\}$ **if** $(cond)$ $\{stmt_1\}$ **else** $\{stmt_2\}$ $\{\mathbf{Q}\}$
  Using the inductive hypothesis, we know that $\{\mathbf{P_1}\}$ $stmt_1$; $\{\mathbf{Q_1}\}$ and $\{\mathbf{P_2}\}$ $stmt_2$; $\{\mathbf{Q_2}\}$ verify their postconditions and result in a valid Store. Breaking each case of the conditional (taken from rule 10):
  - $\mathbf{P} \wedge cond \neq \mathbf{null} \Rightarrow \mathbf{P_1}$ and $\mathbf{Q_1} \Rightarrow \mathbf{Q}$, so the resulting store $\sigma$' (that verifies $\mathbf{Q}$) will also be valid.
  - $\mathbf{P} \wedge cond == \mathbf{null} \Rightarrow \mathbf{P_2}$ and $\mathbf{Q_2} \Rightarrow \mathbf{Q}$, so the resulting store $\sigma$' (that verifies $\mathbf{Q}$) will also be valid.

□

**Definition 12.** *A valid Store is also* ***acyclic*** *when all the references whose class is annotated as acyclic are reachable for only one path (itself). Taking into account that an acyclic class must have only acyclic fields, this definition prevents from having cycles deeper in the reachable paths of the reference.*

**Theorem 3.** *If a store $\sigma$ is valid and acyclic, then for all acyclic reference $o$:*

- $footprint(\sigma, \iota, \iota) == 1$
- *For all* $\iota_1, footprint(\sigma, \iota, \iota_1) \geq footprint(\sigma, \iota.field, \iota_1) \qquad field \in fields(o)$

*Proof.* Let's prove each part of the theorem individually:

- $footprint(\sigma, \iota, \iota) == 1$
  We start by assuming that $footprint(\sigma, \iota, \iota) > 1$. The hypothesis of the theorem states that $\sigma$ is valid, therefore there are more that 1 paths to reach $\iota$ from $\iota$. We also know that $\sigma$ is acyclic, so each acyclic reference must only be reachable from itself. We can then conclude that $o$ is not acyclic. This is a contradiction, because $o$ is acyclic, which comes from assuming that $footprint(\sigma, \iota, \iota) > 1$. Therefore, $footprint(\sigma, \iota, \iota) == 1$.

- for all $\iota_1, footprint(\sigma, \iota, \iota_1) \geq footprint(\sigma, \iota.field, \iota_1) \qquad field \in fields(\iota)$
  Let's assume that $footprint(\sigma, \iota, \iota_1) < footprint(\sigma, \iota.field, \iota_1)$. This would mean that there are less paths from $\iota$ to $\iota_1$ than from $\iota.field$ to $\iota_1$ for any field. However the hypothesis of the theorem states that $\sigma$ is valid, which implies that the paths of a reference include the ones of its fields. Therefore $footprint(\sigma, \iota, \iota_1)$ would be at least equal to the footprint value of its fields; reaching a contradiction that comes form assuming that $footprint(\sigma, \iota, \iota_1) < footprint(\sigma, \iota.field, \iota_1)$. Then $footprint(\sigma, \iota, \iota_1) \geq footprint(\sigma, \iota.field, \iota_1)$.

□

**Theorem 4.** *For all valid and acyclic store $\sigma$, environment env, and blocks $\{\mathbf{P}\}$ Stmt $\{\mathbf{Q}\}$, if:*

- $\sigma, env, this \models \mathbf{P}$, *and*
- $Stmt, \sigma, env, this \rightsquigarrow \sigma', env'$, *and*
- $\{\sigma, \sigma'\}, env', this \models \mathbf{Q}$

*then $\sigma'$ is acyclic.*

*Proof.* This proof is an extension of the Theorem 2 and we will follow a similar approach. The inductive cases have the same demonstration as in Theorem 2, therefore we will focus on the relevant base cases:

- $\{\mathbf{P}\}\ x := \textbf{new C}()\ \{\mathbf{Q}\}$
  The effect of the allocation execution is the following (rule in Section 4.2.1):
  - $\sigma'(\iota, \texttt{Footprint})[\iota] = 1$
    Since the object is being created, it only has 1 reference that points to it (itself). Besides, no other reference points to it. Since the operation only affects that reference, using the inductive hypothesis the rest of the elements of the Store preserve the acyclicity.

  Then we conclude that an allocation execution results in an acyclic Store.

- $\{\mathbf{P}\}\ o.field := x\ \{\mathbf{Q}\}$
  The field write operation execution has the following impact on the Store (rule 4.7):
  - $\sigma(o, field) \neq \textbf{null}$ implies that for all $\iota_1, \iota_2,\ \sigma(\iota_1, \texttt{Footprint})[o] > 0$ implies that
    $\sigma'(\iota_1, \texttt{Footprint})[\iota_2] =$
    $\sigma(\iota_1, \texttt{Footprint})[\iota_2] - \sigma(o, field)(\texttt{Footprint})[\iota_2] + \sigma(x, \texttt{Footprint})[\iota_2]$
    $\sigma(o, field) = \textbf{null}$ implies that for all $\iota_1, \iota_2,\ \sigma(\iota_1, \texttt{Footprint})[o] = 0$ implies that
    $\sigma'(\iota_1, \texttt{Footprint})[\iota_2] = \sigma(\iota_1, \texttt{Footprint})[\iota_2] + \sigma(o, \texttt{Footprint})[\iota_2]$
    This predicate only applies to the references that point to $o$ (including $o$ itself). It states that the footprint must add the references of $x$ and release the once of the previous value of $\sigma(o, field)$ (this does not apply when it is null). Notice that:
    - If we are in the acyclic case, because of Corollary 1 we know that the fails predicate does not hold (rule 4.10) it is guaranteed that $\sigma(x, \texttt{Footprint})[o] = 0$ will still be equal to 1, preserving the acyclicity of the resulting Store
    - Since we are adding the footprint of the new assigned field, we are holding the second part of the acyclicity definition (the footprint of a reference includes the ones of its fields)

  Then we conclude that a field write execution results in an acyclic Store.

- $\{\mathbf{P}\}\ x := o.field\ \{\mathbf{Q}\}$
  The field read operation has no effect over the Store (rule in Figure 4.6). Then, using the inductive hypothesis the resulting Store is acyclic.

- $\{\mathbf{P}\}\ id := e_0.\mathtt{m}(e_1, ..., e_n)\ \{\mathbf{Q}\}$
  Because of Corollary 1 the fails predicate does not hold, therefore we can use the *FootprintInvariant* in order to guarantee the acyclicity of the reference parameters in the execution. For each reference, the execution has the following effect on the Store (rule in Figure 4.9):
  - $(isAcyclicQual(\iota)$ implies that $\sigma'(\iota, \texttt{Footprint})[\iota] = 1)$
    Ensures that the reference is only reached from itself (in the acyclic case).
  - (for all $\iota_1\ Ref,\ \sigma'(\iota, \texttt{Footprint})[\iota_1] \geq \sigma'(\sigma'(\iota, f), \texttt{Footprint})[\iota_1]$ and
    $FootprintInvariant(\sigma', \sigma'(\iota, f)))$
    Ensures that the reference's footprint includes the ones of its fields and verifies it recursively.

  Then we conclude that a method invocation execution results in an acyclic Store.

□

# Chapter 5

# Implementation

In the previous chapter we went over the formal definition of the language we have presented in order to enforce acyclicity. This chapter shows an insight on the implementation concepts and details. The first part of the chapter explains the background aspects that are important to take into account in order to understand the implementation. We start by analyzing how we can implement a static verification system for object-oriented programs, in particular with an introduction to the Boogie programming language[Lei08b]. Since it is important to understand its implementation and features, we describe the whole verification language through grammar and semantics (presented as the Verification Conditions formulas it generates). Since Boogie is a lower level language, it is usually used as an intermediate language. For that reason, we then took Dafny as a reference, which is an open source experimental language created at Microsoft Research that provides the basic support for verifying object-oriented programs. Our implementation is based on Dafny's compiler.

The second part of the chapter explains the implementation of our language. Following the grammar and operational semantics presented in Chapter 4, we implemented the parser and translation to Boogie that will be later passed to the prover. To understand the big picture of how the translation is performed, also Dafny's translation rules must be taken into account. The tool's source code and binaries are available at `http://acycliclanguage.neisen.com.ar`.

Figure 5.1 shows the architectural overview of the verifier implementation. With more or less detail, this chapter will cover most of the modules. The first stage of the pipeline parses the program source code and performs a simple type checking. The resulting syntax tree is then used to make a translation into a verification intermediate language that formalizes the semantics and proof obligations of the program. Using the translated program, we get the logical formulas that finally will be analyzed by the theorem prover, searching for proofs or counterexamples. In particular, our work is located in the *verification language translation*.

This section shows several technical details about Boogie and Dafny that are necessary to understand the whole implementation. In Section 5.2.2 we present our contributions to the translation.

## 5.1 Background

Before analyzing the Implementation Details (in Section 5.2), we should cover some of the related work over which we are building the tool we have created. First of all we will show how we have accomplished the program verification, and then we introduce the experimental language in which we built our prototype.

### 5.1.1 Verifying Object-Oriented Programs using Boogie

One of the standard approaches to satisfy program verification is to transform a program into verification conditions (often referred as VC), which are logical formulas whose validity implies that the program satisfies the formulas under consideration. There are different approaches for that; one proven to be

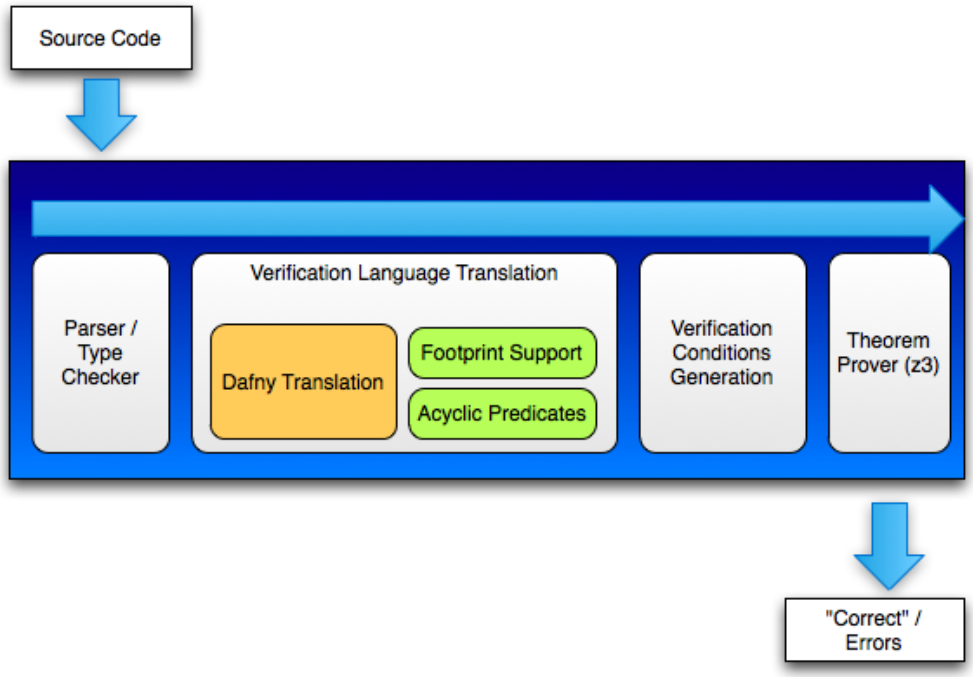



**Figure 5.1:** Architecture of the implementation

successful in practice is the use of an intermediate language since it enables verification of several source languages. There is some work around this [BCD+05] and the choice we have made is to translate the program to BoogiePL[Lei08b].

BoogiePL is an effective intermediate language because it lacks the complexity of an Object-Oriented language and incorporates the logical features needed to generate the Verification Conditions. BoogiePL looks like a high-level assembly language, which consists of a *theory* and an *imperative* part. The theory has declarations of types, constants, functions and axioms. The imperative part has declarations of variables and procedures. A Boogie program has the following form:

$$\begin{aligned} Program \quad ::= \quad & TypeDecl^* \mid SimbolDecl^* \mid VarDeclStmt^* \mid AxiomDecl^* \\ & \mid ProcedureDecl^* \mid ImplementationDecl^* \end{aligned}$$

The theory is declared like this:

$$\begin{aligned} TypeDecl &::= \textbf{type}\ typename; \\ SymbolDecl &::= ConstDecl \mid FunctionDecl \\ ConstDecl &::= \textbf{const}\ var : type; \\ type &::= \textbf{bool} \mid \textbf{int} \mid \textbf{ref} \mid \textbf{name} \mid \textbf{any} \mid typename \mid ArrayType \\ ArrayType &::= [type, type]\ type \\ FunctionDecl &::= \textbf{function}\ function(\ type^*\ )\ \ \textbf{returns}\ \ (\ type\ ); \\ AxiomDecl &::= \textbf{axiom}\ Expr; \end{aligned}$$

BoogiePL is strongly typed in order to improve readability and catch simple errors. However, all that information is erased during the translation into Verification Conditions.

The expressions are formed like this:

$$
\begin{array}{rcl}
Expr & ::= & Literal \mid var \mid UnOp\ Expr \mid Expr\ BinOp\ Expr \mid Expr[Expr, Expr] \\
 & & \mid FuncApp \mid Quant \mid \mathbf{cast}(Expr, type) \mid \mathbf{old}(Expr) \\
Literal & ::= & \mathbf{false} \mid \mathbf{true} \mid \mathbf{null} \mid integer \\
BinOp & ::= & \Leftrightarrow \mid \Rightarrow \mid \vee \mid \wedge \mid <: \mid \leq \mid < \mid \geq \mid > \mid \neq \mid = \mid + \mid - \mid * \mid / \mid \% \\
UnOp & ::= & - \mid \neg \\
FuncApp & ::= & function(\ Expr^*\ ) \\
Quant & ::= & (\forall\ VarDecl^*\ Trigger^*\ \bullet\ Expr\ ) \mid (\exists\ VarDecl^*\ Trigger^*\ \bullet\ Expr\ ) \\
VarDecl & ::= & var\ :\ type\ \langle \mathbf{where}\ Expr \rangle^? \\
Trigger & ::= & \{\ Expr^+\ \}
\end{array}
$$

The $\mathbf{old}(E)$ expression is used in the postconditions and implementations to refer to the value of $E$ in the procedure's pre-state. The **where** clause in a parameter signature forces a constraint, usually referring to the parameter's type. The $Trigger$ is a directive that tells a theorem prover how to instantiate quantifiers.

The imperative part of a BoogiePL program consists of global variable declarations, procedure headers and procedure implementations:

$$
\begin{array}{rcl}
VarDeclStmt & ::= & \mathbf{var}\ \ VarDecl^*; \\
ProcedureDecl & ::= & \mathbf{procedure}\ \ procname\ (\ VarDecl^*\ )\ \langle\ \mathbf{returns}\ \ (VarDecl^*)\rangle^? \\
 & & \langle \mathbf{free}^?\ \mathbf{requires}\ Expr;\rangle^*\ \langle \mathbf{modifies}\ Expr;\rangle^*\ \langle \mathbf{free}^?\ \mathbf{ensures}\ Expr;\rangle^* \\
 & & ImplBody^? \\
ImplementationDecl & ::= & \mathbf{implementation}\ procname\ (\ VarDecl^*\ )\ \langle\ \mathbf{returns}\ \ (VarDecl^*)\rangle^? \\
 & & ImplBody \\
ImplBody & ::= & \{\ VarDeclStmt\ Block^+\ \} \\
Block & ::= & \langle label\ :\rangle^?\ Command^*\ TransferCommand
\end{array}
$$

The clauses marked with **free** mean that the expression is assumed, but not checked. This is useful when encoding properties guaranteed by the language and we can prove that there is no correct code that can violate them.

Finally, the implementation blocks are defined like this:

$$
\begin{array}{rcl}
Command & ::= & Passive \mid Assign \mid Call; \\
Passive & ::= & \mathbf{assert}\ Expr; \mid \mathbf{assume}\ Expr; \\
Assign & ::= & var\ \langle [Expr, Expr]\rangle^?\ :=\ Expr; \mid \mathbf{havoc}\ \ var^+; \\
Call & ::= & \mathbf{call}\ \ var^*\ :=\ procname(Expr^*); \\
TransferCommand & ::= & \mathbf{goto}\ \ label^+; \mid \mathbf{return} \mid IfStmt \mid \mathbf{while}\ (Expr)\ LoopInv^*\ Block; \\
LoopInv & ::= & \mathbf{free}^?\ \mathbf{invariant}\ Expr; \\
IfStmt & ::= & \mathbf{if}\ (Expr)\ Block\ Else^? \\
Else & ::= & \mathbf{else}\ Block \mid \mathbf{else}\ IfStmt
\end{array}
$$

The **havoc** statement assigns its variables to an arbitrary value. The **assume** statement restricts the execution to the Boogie program that satisfies its expression. We need to be careful when using it, because if the expression evaluates to **false**, the execution will be infeasible (ie. it will trivially verify the program). The **assert** statement forces a proof obligation, that when its expression is **false**, the execution results in an unrecoverable error.

The challenge now is finding the right translation from our language into BoogiePL, ensuring that the programs are correctly verified and the acyclicity is guaranteed (when applied).

### 5.1.2 Verification Conditions

One of the last steps when working on static program verification is generating verification conditions from an input program (such as BoogiePL) and passes them to a theorem prover. As we have mentioned earlier, a verification condition is a first-order logical formula whose validity implies that a program satisfies its specification. This task has a great impact on the effectiveness of the verification[FS01].

Following the discussion in [BL05], the transformation applied is *weakest precondition*. The definition of this $\mathsf{wp}[\![.]\!]$ can be also considered as the semantics of BoogiePL.

Weakest precondition ($\mathsf{wp}[\![C, Q]\!]$) is a predicate transformer of sequential imperative programs [Dij97]. That is, given a statement $C$ and postcondition $Q$, it constructs a precondition $P$ such that $\{P\}\ C\ \{Q\}$ is valid and $P$ is the weakest formula that can establish $Q$ as a postcondition for $C$. By *weakest*, it means that the precondition returned from $\mathsf{wp}[\![C, Q]\!]$ imposes the fewest restrictions on the input of $C$ so that it guarantees $Q$. A simple example would be:

$$\mathsf{wp}[\![x := y - 5, x > 10]\!] \ = \ (y - 5 > 10) \ = \ (y > 15)$$

The $\mathsf{wp}[\![.]\!]$ function in this work is defined as[BL05, Lei08a]:

$$\begin{array}{l}
\mathsf{wp}[\![xs := EE;,\ Q]\!] = Q[\![EE/xs]\!] \\
\mathsf{wp}[\![\textbf{havoc}\ xs;,\ Q]\!] = (\forall xs \bullet Q) \\
\mathsf{wp}[\![\textbf{assert}\ E;,\ Q]\!] = E \ \wedge \ Q \\
\mathsf{wp}[\![\textbf{assume}\ E,\ Q]\!] = E \ \Rightarrow \ Q \\
\mathsf{wp}[\![S\ T,\ Q]\!] = \mathsf{wp}[\![S,\ \mathsf{wp}[\![T,\ Q]\!]]\!] \\
\mathsf{wp}[\![\textbf{if}\ (E)\ \{\ S\ \}\ \textbf{else}\ \{\ T\ \},\ Q]\!] = (E \Rightarrow \mathsf{wp}[\![S,\ Q]\!]) \ \wedge \ (\neg E \Rightarrow \mathsf{wp}[\![T,\ Q]\!]) \\
\mathsf{wp}[\![\textbf{while}\ (\ E\ )\ \textbf{invariant}\ J;\ \{\ S\ \},\ Q]\!] = \\
\quad J \ \wedge \ (\forall xs \bullet J \ \wedge \ E \ \Rightarrow \ \mathsf{wp}[\![S,\ J]\!]) \ \wedge \ (\forall xs \bullet J \ \wedge \ \neg E \ \Rightarrow \ Q)
\end{array}$$

The definition of procedure calls and definitions is a bit tricky. The calls must rely on the procedure's specification contract (essential for information hiding and modular verification[Par72]) and the implementations are checked to satisfy its implementation. In the definition, lets consider the following procedure template:

**procedure** $P(ins)$ **returns** $(outs)$;
  **requires** $Pre$;
  **modifies** $mod$;
  **ensures** $Post$;
  $\{\ stmts\ \}$

$$\begin{array}{l}
\mathsf{wp}[\![\textbf{call}\ xs \ := \ P(EE);,\ Q]\!] = \\
\quad \mathsf{wp}[\![ins' := EE; \textbf{assert}\ Pre'; mod' := mod; \textbf{havoc}\ mod, outs'; \textbf{assume}\ Post'; xs := outs';,\ Q]\!] \\
\mathsf{wp}[\![P,\ Q]\!] = \qquad \text{// P is a procedure definition} \\
\quad \mathsf{wp}[\![Axioms \ \Rightarrow \ \mathsf{wp}[\![\textbf{assume}\ Pre; mod' := mod; stmts'; \textbf{assert}\ Post';,\ \textbf{true}]\!],\ Q]\!]
\end{array}$$

More details on the Verification Conditions generation and theorems provers can be found in [Lei08b], [BCD+05] and [dMB08].

### 5.1.3 Dafny language

Dafny [Lei08a] is an experimental language created by K. Rustan M. Leino that explores the dynamic frames style of specifications in an object-based sequential setting. It provides a solid foundation in order to generate the Verification Conditions that serve as input for a solver. The translation uses BoogiePL as the intermediate language. The language implementation is open-source and it can be found at [Mic09b]. For those reasons we decided to use Dafny as the starting point of the implementation of our language. Figure 5.2 shows the grammar of a Dafny program and Figure 5.3 its basic code skeleton. Notice that the grammar is very similar to the one of the language we have defined in Figure 4.1.

It is worth mentioning that the *dynamic frames* are implemented using the $footprint$ variable (which is a set of objects). One of the features that underlie Dafny's specification is the use of ghost fields, which are intended to be used only for verification purposes and should not be included in a compiled program. In Section 2.1 we referred to them as specification variables.

Following the grammar definition, it uses the Coco/R compiler generator[Ter04] in order to create the scanner and parser that will be used to make the translation into BoogiePL discussed later.

$$
\begin{array}{rcl}
\mathit{Program} & ::= & \mathit{Class}^* \\
\mathit{Class} & ::= & \textbf{class}\ \mathit{Id}\ \{\mathit{Member}^*\} \\
\mathit{Member} & ::= & \mathit{Field} \mid \mathit{Method} \\
\mathit{Field} & ::= & \textbf{var}\ \mathit{Id} : \mathit{Type} \\
\mathit{Type} & ::= & \textbf{bool} \mid \textbf{int} \mid \mathit{Id} \mid \textbf{object} \\
\mathit{Method} & ::= & \textbf{method}\ \mathit{Id}(\mathit{Param}^*)\ \textbf{returns}\ (\mathit{Param}^*)\ \mathit{Spec}^*\ \{\ \mathit{Stmt}^*\} \\
\mathit{Param} & ::= & \mathit{Id} : \mathit{Type} \\
\mathit{Stmt} & ::= & \textbf{var}\ x : \mathit{Type}; \\
 & \mid & x := \mathit{Expr}; \\
 & \mid & \mathit{Expr}.f := \mathit{Expr}; \\
 & \mid & x := \mathit{new}\ \mathit{Type}; \\
 & \mid & \textbf{if}\ (\mathit{Expr})\{\mathit{Stmt}^*\}\ \textbf{else}\ \{\mathit{Stmt}^*\} \\
 & \mid & \textbf{while}\ (\mathit{Expr})\ \mathit{Invariant}^*\ \{\mathit{Stmt}^*\} \\
 & \mid & \textbf{call}\ xs := \mathit{Expr}.\mathit{Id}(\mathit{Expr}^*); \\
\mathit{Expr} & ::= & x \mid \textbf{this} \mid \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \mid \mathit{Expr}.f \mid \textbf{old}(\mathit{Expr}) \mid \textbf{fresh}(\mathit{Expr}) \\
 & & \textbf{fresh}(\mathit{Expr}) \mid \forall x \bullet \mathit{Expr} \\
\mathit{Spec} & ::= & \textbf{requires}\ \mathit{Expr}; \mid \textbf{modifies}\ \mathit{Expr}; \mid \textbf{ensures}\ \mathit{Expr}; \\
\mathit{Invariant} & ::= & \textbf{invariant}\ \mathit{Expr};
\end{array}
$$

**Figure 5.2:** Dafny

```
class Coo {
    var footprint : set<object>;
    // other fields go here...
    function Valid() : bool
        reads this, footprint;
    {
        this ∈ footprint // something more substantial goes here...
    }

    method Init()
        modifies this;
        ensures Valid() ∧ fresh(footprint−{this});
    {
        footprint := {this};
        // more initialization goes here...
    }

    method Mutate()
        requires Valid();
        modifies footprint;
        ensures Valid() ∧ fresh(footprint−old(footprint));
    {
        // mutations of the object go here...
    }
}
```

**Figure 5.3:** Dafny code skeleton

### Translation into BoogiePL

One of the most important contributions of Dafny to this work is the basic translation of an Object-Oriented language into BoogiePL. This section presents an overview of Dafny's translation implementation.

Dafny's prelude declares some basic types and functions that will be used across the translation. It also includes the declarations of classes, methods and functions declared in the Dafny program (the result of $\mathsf{Df}[\![.]\!]$ function):

**type** $ClassName$;
**type** $Ref$;
**type** $Set\ \alpha = [\alpha]bool$;
**type** $Seq\ \alpha$;
**const** $null : Ref$;

Classes are translated as follows:

$\mathsf{Decl}[\![\textbf{class}\ C\{members\}]\!] =$
  **const unique** $class.C : ClassName$;
  $\mathsf{Decl}^*[\![members]\!]$

Types are translated using the $\mathsf{Type}[\![.]\!]$ function:

$\mathsf{Type}[\![\textbf{bool}]\!] = \texttt{bool}$
$\mathsf{Type}[\![\textbf{int}]\!] = \texttt{int}$
$\mathsf{Type}[\![Id]\!] = Ref$
$\mathsf{Type}[\![\textbf{set} < T >]\!] = Set\ \mathsf{Type}[\![T]\!]$
$\mathsf{Type}[\![\textbf{seq} < T >]\!] = Seq\ \mathsf{Type}[\![T]\!]$

When modeling an object-oriented semantics in Boogie, it is important to analyze how to model the memory (specially when we want to include dynamic object allocations and references). Some discussions about this can be found at [Lei08a] and [Lei08b]. Dafny's implementation of the heap consists of a map which keys are a reference and a *Field* name, and the value is the field. Then, it declares a global variable (named $\mathcal{H}$) of type Heap. For that reason, the following declarations are added to the prelude:

**type** $Field\ \alpha$
**type** $HeapType = \langle\alpha\rangle[Ref, Field\ \alpha]\alpha$
**var** $\mathcal{H} : HeapType$;
**const unique** $alloc : Field\ \texttt{bool}$

Since the map domain is total, unreferenced objects are valid keys. For that reason, the boolean *alloc* field is introduced, which given a reference returns if it is currently allocated.

As a consequence of the heap implementation, the fields must be encoded into unique names:

$\mathsf{Decl}[\![\textbf{var}\ field : T;]\!] =$
  **const unique** $SomeClass.field : Field\ \mathsf{Type}[\![T]\!]$;

The expressions translation is defined by the $\mathsf{Tr}[\![.]\!]$ function:

$\mathsf{Tr}[\![x]\!] = x$
$\mathsf{Tr}[\![\textbf{this}]\!] = this$
$\mathsf{Tr}[\![E \square F]\!] = \mathsf{Tr}[\![E]\!] \square \mathsf{Tr}[\![F]\!]$
$\mathsf{Tr}[\![E.field]\!] = \mathcal{H}[\mathsf{Tr}[\![E]\!], SomeClass.field]$
$\mathsf{Tr}[\![\textbf{old}(E)]\!] = \textbf{old}(\mathsf{Tr}[\![E]\!])$
$\mathsf{Tr}[\![\textbf{fresh}(E)]\!] = (\forall o : Ref \bullet o \in \mathsf{Tr}[\![E]\!] \Rightarrow o = null \vee \neg\textbf{old}(\mathcal{H})[o, alloc])$
$\mathsf{Tr}[\![\forall x \bullet E]\!] = (\forall x \bullet \mathsf{Tr}[\![E]\!])$

The $\mathsf{Df}[\![.]\!]$ function validates that a Dafny expression is well formed (it will be used when translating the method statements):

$\mathsf{Df}[\![x]\!] = \mathbf{true}$
$\mathsf{Df}[\![\mathbf{this}]\!] = \mathbf{true}$
$\mathsf{Df}[\![E \ \square \ F]\!] = \mathsf{Df}[\![E]\!] \ \wedge \mathsf{Df}[\![F]\!]$
$\mathsf{Df}[\![E.field]\!] = \mathsf{Df}[\![E]\!] \wedge \mathsf{Tr}[\![E]\!] \neq null$
$\mathsf{Df}[\![\mathbf{old}(E)]\!] = \mathbf{old}(\mathsf{Df}[\![E]\!])$
$\mathsf{Df}[\![\mathbf{fresh}(E)]\!] = \mathsf{Df}[\![E]\!]$
$\mathsf{Df}[\![\forall x \bullet E]\!] = \mathsf{Df}[\![E]\!]$

The translation of the method $Foo$ in $SomeClass$ is defined like this:

$\mathsf{Decl}[\![\mathbf{method}\ Foo(ins)\ \mathbf{returns}\ (outs)\mathbf{requires}\ Pre; \mathbf{modifies}\ Mod; \mathbf{ensures}\ Post; \{stmts\}]\!] =$
  $\mathbf{procedure}\ SomeClass.Foo(this : Ref, \mathsf{Decl}^*[\![ins]\!])\ \ \mathbf{returns}\ \ (\mathsf{Decl}^*[\![outs]\!])$
    $\mathbf{free\ requires}\ this \neq null \wedge \mathsf{GoodRef}[\![this, SomeClass, \mathcal{H}]\!];$
    $\mathbf{free\ requires}\ \mathsf{IsAllocated}^*[\![ins]\!];$
    $\mathbf{requires}\ \mathsf{Tr}[\![Pre]\!];$
    $\mathbf{modifies}\ \mathcal{H};$
    $\mathbf{free\ ensures}\ \mathsf{IsAllocated}^*[\![outs]\!];$
    $\mathbf{ensures}\ \mathsf{Tr}[\![Post]\!];$
    $\mathbf{free\ ensures}\ (\forall o : Ref, f : Field\ \alpha)$
      $(\mathcal{H}[o, f] = \mathbf{old}(\mathcal{H})[o, f] \vee (o \in \mathbf{old}(\mathsf{Tr}[\![Mod]\!]) \vee \neg\mathbf{old}(\mathcal{H})[o, alloc]);$
    $\mathbf{free\ ensures}\ (\forall o : Ref)(\mathbf{old}(\mathcal{H})[o, alloc] \Rightarrow \mathcal{H}[o, alloc]);$
  {
    $\mathbf{var}\ \mathsf{Locals}^*[\![stmts]\!];$
    $\mathsf{Stmt}^*_{Mod}[\![stmts]\!];$
  }

$\mathsf{Decl}[\![x : T]\!] = x : \mathsf{Type}[\![T]\!]$
$\mathsf{IsAllocated}[\![x : T]\!] = T$ is reference type $\Rightarrow \mathsf{GoodRef}[\![x, T, \mathcal{H}]\!]$

$$\mathsf{GoodRef}[\![t, T, h]\!] = \begin{cases} t = null \vee (h[t, alloc] \wedge dtype(t) = class.T) & \text{if } T \text{ is a class name} \\ t = null \vee h[t, alloc] & \text{if } T \text{ is } \mathbf{object} \end{cases}$$

Notice that the translation makes the heap be modified in all the methods. The framing is controlled by adding some postconditions at the end of the method contract.

For the translation of methods, we need to be aware that in Boogie all the local variables must be declared at the beginning of the procedure. The $\mathsf{Locals}[\![.]\!]$ function extracts the local variables defined in each expression:

$\mathsf{Locals}[\![\mathbf{var}\ x : T]\!] = x : \mathsf{Type}[\![T]\!]$
$\mathsf{Locals}[\![x := E]\!] = //$ Empty
$\mathsf{Locals}[\![E.f := E]\!] = //$ Empty
$\mathsf{Locals}[\![x := \mathbf{new}\ T]\!] = //$ Empty
$\mathsf{Locals}[\![\mathbf{assert}\ E]\!] = //$ Empty
$\mathsf{Locals}[\![\mathbf{assume}\ E]\!] = //$ Empty
$\mathsf{Locals}[\![\mathbf{if}\ (E)\{S_0\}\ \mathbf{else}\ \ \{S_1\}]\!] = \mathsf{Locals}[\![S_0]\!], \mathsf{Locals}[\![S_1]\!]$
$\mathsf{Locals}[\![\mathbf{call}\ xs := E.M(E_1)]\!] = //$ Empty
$\mathsf{Locals}[\![\mathbf{while}\ (E)\ invs\ \{S\}]\!] = prevHeap, \mathsf{Locals}[\![S]\!]$

And the statements are translated using the $\mathsf{Stmt}_{Mod}[\![.]\!]$ function:

$\mathsf{Stmt}_{Mod}[\![\mathbf{var}\ x : T;]\!] =$
  $\mathbf{havoc}\ x;$
$\mathsf{Stmt}_{Mod}[\![x := E;]\!] =$
  $\mathbf{assert}\ \mathsf{Df}[\![E]\!];$
  $x := \mathsf{Tr}[\![E]\!];$
$\mathsf{Stmt}_{Mod}[\![E_0.f := E_1;]\!] =$
  $\mathbf{assert}\ \mathsf{Df}[\![E_0]\!] \wedge \mathsf{Df}[\![E_1]\!];$

**assert** $\mathsf{Tr}[\![E_0]\!] \in \mathbf{old}(\mathsf{Tr}[\![Mod]\!]) \vee \neg\mathbf{old}(\mathcal{H})[\mathsf{Tr}[\![E_0]\!], alloc]$;
$\mathcal{H}[\mathsf{Tr}[\![E_0]\!], C.f] := \mathsf{Tr}[\![E_1]\!]$;
$\mathsf{Stmt}_{Mod}[\![x := \mathbf{new}\ T;]\!] =$
**havoc** $x$; **assume** $x \neq null \wedge \neg\mathcal{H}[x, alloc] \wedge dtype(x) = T$;
$\mathcal{H}[x, alloc] := \mathbf{true}$;
$\mathsf{Stmt}_{Mod}[\![\mathbf{assert}\ E;]\!] =$
**assert** $\mathsf{Df}[\![E]\!]$;
**assert** $\mathsf{Tr}[\![E]\!]$;
$\mathsf{Stmt}_{Mod}[\![\mathbf{if}\ (E)\{S_0\}\ \mathbf{else}\ \{S_1\}]\!] =$
$\mathsf{Df}[\![E]\!]$;
**if** $(\mathsf{Tr}[\![E]\!])\{\mathsf{Stmt}_{Mod}[\![S_0]\!]\}$ **else** $\{\mathsf{Stmt}_{Mod}[\![S_1]\!]\}$;
$\mathsf{Stmt}_{Mod}[\![\mathbf{while}\ (E)\ \mathbf{invariant}\ \ J; \{S\}]\!] =$
$prevHeap := \mathcal{H}$;
**while** $(\mathsf{Tr}[\![E]\!])$
**invariant** $\mathsf{Df}[\![J]\!] \wedge \mathsf{Tr}[\![J]\!]$;
**invariant** $\mathsf{Df}[\![E]\!]$;
**free ensures** $(\forall o : Ref, f : Field\ \alpha)$
$(prevHeap[o, f] = \mathbf{old}(prevHeap)[o, f] \vee$
$(o \in \mathbf{old}(\mathsf{Tr}[\![Mod]\!]) \vee \neg\mathbf{old}(prevHeap)[o, alloc])$;
**free ensures** $(\forall o : Ref)(\mathbf{old}(prevHeap)[o, alloc] \Rightarrow prevHeap[o, alloc])$;
$\{\mathsf{Stmt}^*_{Mod}[\![S]\!]\}$
$\mathsf{Stmt}_{Mod}[\![\mathbf{call}\ \ xs := E.M(EE);]\!] =$
**assert** $\mathsf{Df}[\![E]\!] \wedge \mathsf{Df}^*[\![EE]\!] \wedge \mathsf{Tr}[\![E]\!] \neq null$;
**assert** $(\forall o : Ref)(o \in \mathsf{Tr}[\![\mathbf{modifies}\ _{C.M}[\![EE/args_{C.M}]\!]]\!] \Rightarrow$
$(o \in \mathbf{old}(\mathsf{Tr}[\![Mod]\!]) \vee \neg\mathbf{old}(\mathcal{H})[o, alloc]))$;
**call** $xs := C.M(\mathsf{Tr}[\![E]\!], \mathsf{Tr}^*[\![EE]\!])$;

Finally, since all Dafny's functions are pures, the translation produces an axiom as follows:

$\mathsf{Decl}[\![\mathbf{function}\ F(ins) : T\ \mathbf{requires}\ R; \mathbf{reads}\ rd; \{body\}]\!]$
**axiom** $(\forall \mathcal{H} : HeapType, this : Ref, \mathsf{Decl}^*[\![ins]\!])$
$(this \neq null \wedge \mathsf{Df}[\![R]\!] \wedge \mathsf{Tr}[\![R]\!] \Rightarrow C.F(\mathcal{H}, this, ins) = \mathsf{Tr}[\![body]\!])$

## 5.2 Implementation Details

Since Dafny's implementation is open-source [Mic09b], we decided to build in top of it all the concepts that we have presented in Chapter 4. This section describes how we accomplish that goal.

### 5.2.1 Footprint

One of the decisions for the language that we made was to provide an *implicit* support for a footprint, like the one presented in Section 3.1. Since the footprint must be available for all the references, we decided to store it in the $\mathcal{H}$ already defined in Dafny. For that reason, we included the following statements in the prelude:

**const unique** $Footprint : Field\ [ref]int$;

And we include two axioms to enforce the valid state of the footprint. The checks added ensure that the footprint value is always non-negative (a negative footprint value does not have any meaning) and that when an object is reached (or its footprint value is positive), it is allocated:

**axiom** $(\forall \mathcal{H} : HeapType, o : Ref, t : Ref :: \{\mathcal{H}[o, Footprint][t]\})$
$(o \neq null \Rightarrow \mathcal{H}[o, Footprint][t] \geq 0)$;
**axiom** $(\forall \mathcal{H} : HeapType, o : Ref, t : Ref :: \{\mathcal{H}[o, Footprint][t]\})$
$(o \neq null \wedge \mathcal{H}[o, alloc] \wedge \mathcal{H}[o, Footprint][t] > 0 \Rightarrow t \neq null \wedge \mathcal{H}[t, alloc])$;

### 5.2.2 Additions to the Translation

This section introduces all the relevant work that we have done in the translation functions in order to implement the acyclicity features that we are presenting. We will also show the traceability between the implementation decisions, and the static semantics and failing executions defined in Sections 4.4 and 4.3.

#### Classes

When translating a class declaration, we need to include the implementation of the class footprint invariant (from Definition 2 of Chapter 3). Notice that the implementation is almost the same as the one defined in the static semantics.

$$
\begin{array}{l}
\mathsf{Decl}[\![\{attrs\}\textbf{class } C\{members\}]\!] = \\
\quad \textbf{const unique } class.C : ClassName; \\
\quad \mathsf{Decl}^*[\![members]\!] \\
\quad \textbf{function } FootprintInvariant\_C(Heap : HeapType, this : Ref) \textbf{ returns } (\texttt{bool}); \\
\quad \textbf{axiom } (\forall Heap : HeapType, this : Ref :: \{FootprintInvariant\_C(Heap, this)\}) \\
\qquad (this \neq null \Rightarrow FootprintInvariant\_C(Heap, this) = \\
\qquad\quad ((\mathsf{IsAcyclic}[\![attrs]\!] \Rightarrow Heap[this, Footprint][this] = 1) \wedge \\
\qquad\quad (\neg \mathsf{IsAcyclic}[\![attrs]\!] \Rightarrow Heap[this, Footprint][this] >= 1) \wedge \\
\qquad\quad Heap[this, Footprint][null] = 0 \wedge \mathsf{FieldInv}^*_{Heap}[\![this, \mathsf{IsAcyclic}[\![attrs]\!], members]\!]));
\end{array}
$$

$$
\mathsf{IsAcyclic}[\![attrs]\!] = \text{acyclic} \in attrs
$$

$$
\begin{array}{l}
\mathsf{FieldInv}_{Heap}[\![this, isAcyclic, member]\!] = \\
\quad \begin{cases}
Heap[this, C.field] \neq null \Rightarrow & \text{if } member \text{ is} \\
\quad Heap[this, Footprint][Heap[this, C.field]] \geq 1 \wedge & \textbf{var } field : T \\
\quad (isAcyclic \Rightarrow Heap[Heap[this, C.field], Footprint][this] = 0 \wedge & \\
\quad Heap[Heap[this, C.field], Footprint] \leq Heap[this, Footprint]) \wedge & \\
\quad FootprintInvariant\_T(Heap, Heap[this, C.field]) & \\
\textbf{true} & \text{otherwise}
\end{cases}
\end{array}
$$

We are also ensuring that the class is well formed with regards to its acyclic structure. Remember from Definition 2 that an acyclic class must have all its fields acyclic too:

$$
\mathsf{Df}[\![\{attrs\}\textbf{class } C\{members\}]\!] = \mathsf{IsAcyclic}[\![attrs]\!] \Rightarrow \mathsf{IsAcyclicField}^*[\![members]\!]
$$

$$
\mathsf{IsAcyclicField}[\![member]\!] = \begin{cases}
\mathsf{IsAcyclic}[\![attrs_T]\!] & \text{if } member \text{ is } \textbf{var } field : T \\
\textbf{true} & \text{otherwise}
\end{cases}
$$

#### Allocation

When a new instance is being allocated, we need to set all its fields to *null* and specify the effect of that statement in all the footprints.

$$
\begin{array}{l}
\mathsf{Stmt}_{Mod}[\![x := \textbf{new } T;]\!] = \\
\quad \oplus \textbf{havoc } x; \ \textbf{assume } x \neq null \wedge \neg \mathcal{H}[x, alloc] \wedge dtype(x) = T; \\
\quad \star \textbf{assume } (\forall m : Ref :: \{\mathcal{H}[x, Footprint][m]\})(x \neq m \Rightarrow \mathcal{H}[x, Footprint][m] = 0) \wedge \\
\qquad (\forall n : Ref :: \{\mathcal{H}[n, Footprint][x]\})(x \neq n \Rightarrow \mathcal{H}[n, Footprint][x] = 0) \wedge \\
\qquad \mathcal{H}[x, Footprint][x] = 1; \\
\quad \mathsf{InitField}^*[\![x, fields_T]\!] \\
\quad \mathcal{H}[x, alloc] := \textbf{true};
\end{array}
$$

$$
\mathsf{InitField}[\![x, field]\!] = \mathcal{H}[x, field] := null;
$$

Remembering the allocation operation static semantics definition, we are only implementing its post-condition (rule 7):

$\oplus$ is to enforce the $new(\mathcal{S}_{pre}, \mathcal{S}_{pos}, x)$ property

$\star$ is the the translation of $footprint(\mathcal{S}_{pos}, x, x) == \bar{1} \wedge (\forall ref_1 : Ref \bullet ref_1 \neq x \Rightarrow$
$footprint(\mathcal{S}_{pos}, x, ref_1) == \bar{0} \wedge footprint(\mathcal{S}_{pos}, ref_1, x) == \bar{0}) \wedge$
$(\forall ref_1, ref_2 : Ref \bullet ref_1 \neq x \wedge ref_2 \neq x \Rightarrow$
$footprint(\mathcal{S}_{pre}, ref_1, ref_2) == footprint(\mathcal{S}_{pos}, ref_1, ref_2))$

### Assignment

The specification of the Boogie command $\mathcal{H}[E_0, C.f] := E_1$ is not enough to enforce the acyclicity verification when executing an assignment (introduced in Section 3.1). For that reason, we are translating the assignment statement like this:

$\mathsf{Stmt}_{Mod}[\![E_0.f := E_1;]\!] =$
**assert** $\mathsf{Df}[\![E_0]\!] \wedge \mathsf{Df}[\![E_1]\!]$;
**assert** $\mathsf{Tr}[\![E_0]\!] \in$ **old**$(\mathsf{Tr}[\![Mod]\!]) \vee \neg$**old**$(\mathcal{H})[\mathsf{Tr}[\![E_0]\!], alloc]$;
$\odot$**assert** $\mathsf{IsAcyclic}[\![attrs_C]\!] \Rightarrow \mathcal{H}[E_1, Footprint][E_0] = 0$;
**call** $assignField(E_0, C.f, E_1, \mathsf{IsAcyclic}[\![attrs_C]\!])$;
$\otimes$**assume** $\mathcal{H}[E_0, C.f] = E_1$;

And the *assignField* procedure is added in the prelude:

**procedure** $assignField(this : Ref, field : Field\ Ref, data : Ref, acyclic :$ `bool`$)$;
**requires** $this \neq null$;
$\mp$**requires** $data = null \vee \neg acyclic \vee \mathcal{H}[data, Footprint][this] = 0$;
**modifies** $\mathcal{H}$;
**ensures** $\mathcal{H}[this, field] = data$;
$\ddagger$**ensures** (**old**$(\mathcal{H})[this, field] \neq null \Rightarrow ((\forall ref_1 : Ref, ref_2 : Ref :: \{\mathcal{H}[ref_1, Footprint][ref_2]\})$
$(\mathcal{H}[ref_1, Footprint][this] > 0 \Rightarrow \mathcal{H}[ref_1, Footprint][ref_2] =$
**old**$(\mathcal{H})[ref_1, Footprint][ref_2] -$ **old**$(\mathcal{H})[$**old**$(\mathcal{H})[this, field], Footprint][ref_2] +$
$\mathcal{H}[data, Footprint][ref_2])$;
$\ddagger$**ensures** (**old**$(\mathcal{H})[this, field] = null \Rightarrow ((\forall ref_1 : Ref, ref_2 : Ref :: \{\mathcal{H}[ref_1, Footprint][ref_2]\})$
$(\mathcal{H}[ref_1, Footprint][this] > 0 \Rightarrow \mathcal{H}[ref_1, Footprint][ref_2] =$
**old**$(\mathcal{H})[ref_1, Footprint][ref_2] + \mathcal{H}[data, Footprint][ref_2])$;
$\ddagger$**ensures** $(\forall ref_1 : Ref, ref_2 : Ref :: \{\mathcal{H}[ref_1, Footprint][ref_2]\})$
$(\mathcal{H}[ref_1, Footprint][this] = 0 \Rightarrow \mathcal{H}[ref_1, Footprint][ref_2] =$ **old**$(\mathcal{H})[ref_1, Footprint][ref_2])$;

The assignment implementation can be traceable with the static semantics as follows (rule 9):

$\odot\mp$ enforce that **requires** $o \neq$ **null**$\wedge((isAcyclic(o) \wedge x \neq$ **null**$) \Rightarrow isAcyclic(x) \wedge footprint(\mathcal{S}_{pre}, x, o) == \bar{0})$;. It is worth mentioning that the parser takes care syntactically that $x$ is acyclic (when it applies).

$\otimes$ is the translation of **ensures** $x == \mathcal{S}_{pos}(o.field)$;

$\ddagger$ is the translation of $(\forall ref_1, ref_2 : Ref \bullet (footprint(\mathcal{S}_{pre}, ref_1, o) == \bar{0} \Rightarrow$
$footprint(\mathcal{S}_{pre}, ref_1, ref_2) == footprint(\mathcal{S}_{pos}, ref_1, ref_2)) \wedge$
$(footprint(\mathcal{S}_{pre}, ref_1, o) > \bar{0} \Rightarrow footprint(\mathcal{S}_{pos}, ref_1, ref_2) ==$
$footprint(\mathcal{S}_{pre}, ref_1, ref_2) \dot{+} footprint(\mathcal{S}_{pre}, x, ref_2) \dot{-}$
$footprint(\mathcal{S}_{pre}, \mathcal{S}_{pos}(o.field), ref_2)))$

### Methods

One of the goals of our work is to reduce the specification explosion that may occur when verifying acyclicity. Considering that we are generating a footprint invariant and the operations verify it, we can

improve the method's declaration as follows:

$$\begin{array}{l}
\mathsf{Decl}[\![\textbf{method } Foo(ins) \textbf{ returns } (outs)\textbf{requires } Pre; \textbf{modifies } Mod; \textbf{ensures } Post; \{stmts\}]\!] = \\
\quad \textbf{procedure } SomeClass.Foo(this : Ref, \mathsf{Decl}^*[\![ins]\!]) \textbf{ returns } (\mathsf{Decl}^*[\![outs]\!]) \\
\qquad \textbf{free requires } this \neq null \wedge \mathsf{GoodRef}[\![this, SomeClass, \mathcal{H}]\!]; \\
\qquad \textbf{free requires } \mathsf{IsAllocated}^*[\![ins]\!]; \\
\qquad \oplus\textbf{free requires } \mathsf{RefInvariant}^*[\![ins]\!]; \\
\qquad \oplus\textbf{requires } \mathsf{Tr}[\![Pre]\!]; \\
\qquad \textbf{modifies } \mathcal{H}; \\
\qquad \textbf{free ensures } \mathsf{IsAllocated}^*[\![outs]\!]; \\
\qquad \odot\textbf{ensures } \mathsf{Tr}[\![Post]\!]; \\
\qquad \odot\textbf{free ensures } (\forall o : Ref, f : Field\ \alpha) \\
\qquad\quad (\mathcal{H}[o, f] = \textbf{old}(\mathcal{H})[o, f] \vee (o \in \textbf{old}(\mathsf{Tr}[\![Mod]\!]) \vee \neg\textbf{old}(\mathcal{H})[o, alloc]); \\
\qquad \odot\textbf{free ensures } (\forall o : Ref)(\textbf{old}(\mathcal{H})[o, alloc] \Rightarrow \mathcal{H}[o, alloc]); \\
\qquad \odot\textbf{free ensures } \mathsf{RefInvariant}^*[\![ins]\!]; \\
\qquad \odot\textbf{free ensures } \mathsf{RefInvariant}^*[\![outs]\!]; \\
\quad \{ \\
\qquad \textbf{var } \mathsf{Locals}^*[\![stmts]\!]; \\
\qquad \mathsf{Stmt}^*_{Mod}[\![stmts]\!]; \\
\quad \}
\end{array}$$

$$\mathsf{RefInvariant}[\![ref]\!] = \begin{cases} ref \neq null \Rightarrow FootprintInvariant\_\texttt{dtype}(\texttt{ref})(\mathcal{H}, ref) & \text{if } dtype(ref) \text{ is classtype} \\ \textbf{true} & \text{otherwise} \end{cases}$$

The implementation is based on the semantics (rule 6) as follows:

$\oplus$ enforces that $\forall \mathcal{S}_{pre} \bullet \mathbf{P}(\mathcal{S}_{pre}) \Rightarrow (Pre[y/this, val_i/p_i](\mathcal{S}_{pre}) \wedge$
$FootprintInvariant(\mathcal{S}_{pre}, y) \wedge FootprintInvariant(\mathcal{S}_{pre}, val_i))$

$\odot$ enforces that of $\forall \mathcal{S}_{pre}, \mathcal{S}_{pos} \bullet \exists xs' \bullet (\mathbf{P}[xs'/xs](\mathcal{S}_{pre}) \wedge$
$HeapSucc(\mathcal{S}_{pre}, \mathcal{S}_{pos}, Mod[xs'/xs, y/this, val_i/p_i]) \wedge$
$Pos[y/this, val_i/p_i, xs'/xs, xs/ret](\mathcal{S}_{pre}, \mathcal{S}_{pos}) \wedge alive(xs, \mathcal{S}_{pos}) \wedge$
$FootprintInvariant(\mathcal{S}_{pos}, y) \wedge FootprintInvariant(\mathcal{S}_{pos}, val_i)) \Rightarrow \mathbf{Q}(\mathcal{S}_{pre}, \mathcal{S}_{pos})$

**Macros**

The macros defined to reduce some of the verification clauses are:

$$\begin{array}{l}
\mathsf{Tr}[\![\textbf{modifies } footprintsWith(refs)]\!] = \\
\quad \textbf{free ensures } (\forall ref_1 : Ref, ref_2 : Ref :: \textbf{old}(\mathcal{H})[ref_1, alloc] \wedge \\
\qquad \bigvee_{o \in refs} \textbf{old}(\mathcal{H})[ref_1, Footprint][o] = 0 \Rightarrow \mathcal{H}[ref_1, Footprint][ref_2] = \\
\qquad\quad \textbf{old}(\mathcal{H})[ref_1, Footprint][ref_2]);
\end{array}$$

$$\begin{array}{l}
\mathsf{Tr}[\![JoinFootprints(base, connected)]\!] = \\
\quad \textbf{free ensures } (\forall ref_1 : Ref, ref_2 : Ref :: \textbf{old}(\mathcal{H})[ref_1, alloc] \wedge \\
\qquad \textbf{old}(\mathcal{H})[ref_1, Footprint][base] > 0 \Rightarrow \mathcal{H}[ref_1, Footprint][ref_2] = \\
\qquad\quad \textbf{old}(\mathcal{H})[ref_1, Footprint][ref_2] + \textstyle\sum_{o \in connected} \mathcal{H}[o, Footprint][ref_2]);
\end{array}$$

$$\begin{array}{l}
\mathsf{Tr}[\![Unreachable(base, target)]\!] = \\
\quad \mathcal{H}[base, Footprint][target] = 0
\end{array}$$

## 5.3 Translation Example

This section will present a simple example of the result of the translation of a method that modifies an object structure. The input program is the following:

```
class {:acyclic} Node {
    var next : Node;
}

class Main {
    method add(a : Node, b : Node)
        requires a != null && b != null && a != b;
        requires Unreachable(b, a);
        modifies a, footprintsWith({a});
        ensures a.next == b;
        ensures JoinFootprint(a, b);
        ensures a.footprint[b] >= 1;
    {
        a.next := b;
    }
}
```

Let's analyze the translation of the add method. The resulting Boogie procedure definition is:

```
1  procedure Main.add(this: ref where this != null && Heap[this, alloc] &&
2      dtype(this) == class.Main, a: ref where
3      Heap[a, alloc] && dtype(a) == class.Node, b: ref where
4      Heap[b, alloc] && dtype(b) == class.Node);
5    free requires a != null ==> FootprintInvariant_Node(Heap, a);
6    free requires b != null ==> FootprintInvariant_Node(Heap, b);
7    // user-defined preconditions
8    requires a != null && b != null && a != b;
9    requires Heap[b, Footprint][a] == 0;
10   modifies Heap;
11   free ensures a != null ==> FootprintInvariant_Node(Heap, a);
12   free ensures b != null ==> FootprintInvariant_Node(Heap, b);
13   ensures Heap[a, Node.next] == b;
14   free ensures (forall o: ref, m: ref :: old(Heap)[o, Footprint][a]
15     > 0 ==> Heap[o, Footprint][m] == old(Heap)[o, Footprint][m] +
16     Heap[b, Footprint][m]);
17   ensures Heap[a, Footprint][b] >= 1;
18   // Macro: footprintsWith
19   free ensures (forall o: ref, m: ref :: old(Heap)[o, alloc] &&
20     old(Heap)[o, Footprint][a] == 0 ==> Heap[o, Footprint][m]
21     == old(Heap)[o, Footprint][m]);
22   // frame condition
23   ensures (forall<alpha> o: ref, f: Field alpha :: { Heap[o, f] }
24     o != null && old(Heap)[o, alloc] && f != Footprint ==>
25     Heap[o, f] == old(Heap)[o, f] || o == a);
26   // boilerplate
27   free ensures HeapSucc(old(Heap), Heap);
```

- Lines 5, 6, 11 and 12 enforce the *FootprintInvariant*
- Line 9 requires that $b$ does not reach $a$ in order to prevent a cycle when executing the assignment
- Lines 19 to 25 specify the effect over the resulting footprint, using the macros provided by the language

Finally, the method implementation is:

```
1 implementation Main.add(this: ref, a: ref, b: ref)
2 {
3     // ----- assignment statement -----
4     assert a != null;
5     assert Heap[b, Footprint][a] == 0;
6     call assignField(a, Node.next, b, true);
7     assume Heap[a, Node.next] == b;
8     assume IsGoodHeap(Heap);
9 }
```

- Line 5 checks that $b$ does not reach $a$ before executing the assignment (in Line 6)
- Line 6 calls the *assignField* procedure (defined in Subsection 5.2.2), which specifies the impact of the assignment in the state (including the footprint)

## 5.4 Experiments

The last stage of this thesis is the empirical evaluation of the implementation tool. For that, we elaborated a set of sample programs to experiment our language with real world examples. We will not show the source code of most of the samples because of their length, but the whole suite is available at http://acyclicclanguage.neisen.com.ar/experiments. The test cases are the following:

1. A simple acyclic linked list
2. An acyclic set implemented over a balanced binary tree
3. An acyclic dictionary implemented over the set from Item 2

The first experiment is presented in Figure 5.4. The linked list implementation is the one explained in Section 5.3, but we extended it with the main method. This last method creates three nodes and uses the add method to connect them.

The linked list sample code is very *clean*, considering that we did not add many predicates about acyclicity or footprints. After running the verifier, we get a successful verification:


Now, in Figure 5.5 we modify the main method to produce a cycle adding the **call** add(n3, n1) statement.

After running the verifier, we get the following error:

```
class {:acyclic} Node {
    var next : Node;
}

class Client {
    method add(a : Node, b : Node)
        requires a != null && b != null && a != b;
        requires Unreachable(b, a);
        modifies a, footprintsWith({a});
        ensures a.next == b;
        ensures JoinFootprint(a, b);
        ensures a.footprint[b] >= 1;
    {
        a.next := b;
    }

    method main()
    {
        var n1 : Node;
        var n2 : Node;
        var n3 : Node;

        n1 := new Node;
        n2 := new Node;
        n3 := new Node;

        call add(n1, n2);
        call add(n2, n3);
    }
}
```

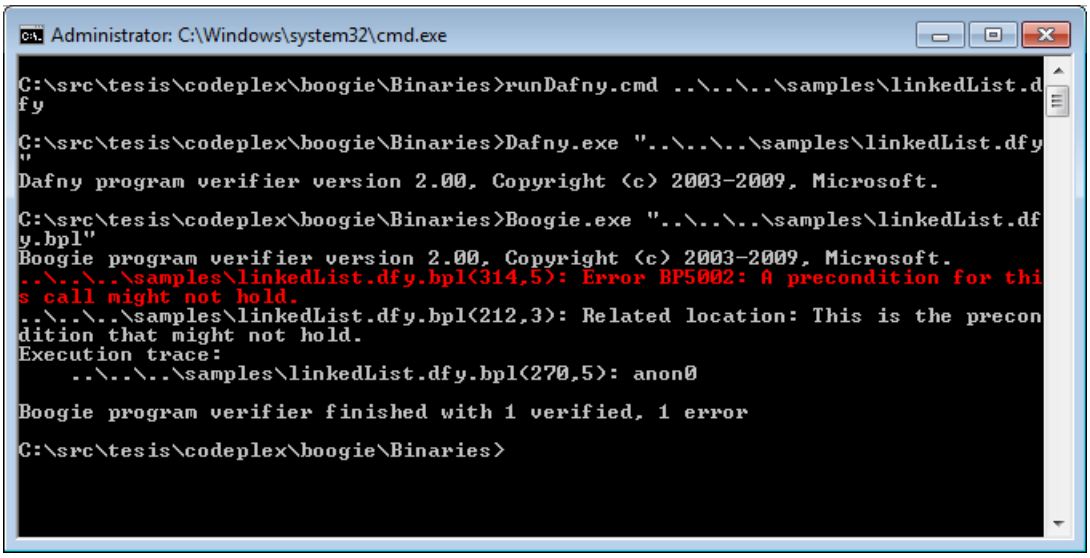
**Figure 5.4:** Linked list experiment

```
    method main()
    {
        var n1 : Node;
        var n2 : Node;
        var n3 : Node;

        n1 := new Node;
        n2 := new Node;
        n3 := new Node;

        call add(n1, n2);
        call add(n2, n3);
        call add(n3, n1);
    }
```

**Figure 5.5:** Linked list experiment producing a cycle




The error shown in the console refers to a *bpl* file, which is the location of the Boogie translation from the program original source code. The verifier indicates the precise Boogie line that fails and our translation links each Boogie statement to the original source code line that it represents. Then, it is easy to trace which line of the program is making the verification fail.

The next experiment is an implementation of an acyclic set over a balanced binary tree. The program consists of the TreeNode and SetBBT classes. The TreeNode exposes its fields (right and left nodes) and the SetBBT declares two methods (one adds an element and the other one is the inclusion). To successfully verify the whole programs, we needed to write some complex predicates and, in consequence, the program has 180 lines of code.

We had a similar experience when implementing an acyclic dictionary over the set from the previous experiment. The Dictionary has method to add an entry and another one to retrieve the definition (by a key). Recalling that one of our goals was to verify modular programs (from Chapter 1), we assumed the validity of the set's specification and developed the dictionary using it. Again, to verify the whole program successfully, we needed to write 100 lines of code.

## 5.5 Lessons Learned

We have learned several lessons from the analysis of the results of the experiments from Section 5.4. We examined the programming experience and the effort needed to successfully verify a program with acyclic data structures.

One of the positive lessons learned is that we could verify the acyclic data structures that we intended. The experiments show that our theoretical ideas are correct, and the tool can be used to create more examples and work on future projects. Moreover, in simple programs we observed that the amount is extra annotations in the contracts is small, since the semantic rules take care of the necessary proof obligations.

The more complex experiments added some difficulties; in general we had to decipher the verifier's error messages to figure out the right predicate to add or change. Also, finding the right loops invariant was quite difficult because we had to include predicates about the footprints. The lesson learned here is that to achieve a successful verification of complex programs, the programmer must have some skills in specifying algorithms and contracts. Leino reported a similar experience when he wrote the Schorr-Waite algorithm in Dafny[Lei10]. In the end, he comments that this task (specifying the algorithm) is not yet for non-experts.

The experience of programing modular programs was also a bit tedious. The modular composition of classes adds some proof obligations needed in the methods contracts. In consequence, we lose some

precision in the footprints after a method invocation. The language semantics (like the footprint invariant) and the macros we propose help to reduce the annotations for the specification of the footprint changes. A potential solution would be to perform inline expansions[Ser97] of small methods.

# Chapter 6

# Conclusions and future work

We started the work in this thesis researching on the design of modular languages, with the goal of incorporating new features that can allow programmers to write better programs with formal guarantees. There are a number of possible ideas in this area and we decided to ask how a language can guarantee that a set of memory references are acyclic.

After analyzing the related work in the subject and discussing about acyclicity, we presented a *toy* language that enforces it, defining the syntax and operational semantics. The language supports method specification with pre/post condition contracts using first order formulas. Then we defined the static semantics that verifies the programs and we prove that a successful verification implies that the program is sound and does not fail.

Finally, we implemented a prototype tool to present our theoretical ideas in a real environment. The tool is a verifier that translates the programs into Boogie[Lei08b] and analyzes the result with the Z3 solver[dMB08]. Using it, we were able to make some experiments and study the results. The lessons learned were both positive and negative. From the positive side, we were able to confirm that our theories are correct by implementing several *acyclic* data structures from the real world. On the other hand, we experienced that as the programs we write get bigger, the annotation burden becomes more tedious and complex. Even though the language semantics and the use of macros reduce the number of annotations, some experiments needed complex method contracts and proof obligations in the code. A future project of work can try to reduce the proof obligations by inferring the effects in the footprints with regards to the verifications written by the programmer. Another useful technique is to produce the method preconditions based on the statements it executes. Doing experiments with Separation Logic[PB08] may deliver some interesting results too.

Most of the work that we have done in this thesis tries to deal with the problem of verifying modular object oriented programs and their side effects. We focused on the study of verifying acyclicity properties of objects over a subset of a Java based language, but we left some open issues for future work. A first enhancement to our language could be the support of inheritance. Since the language semantics assumes static binding, it must be changed to support dynamic binding ([SJPS08] presents a solution to that problem). Besides, dealing with invariant in a modular way is difficult with inheritance[BCD+05]. Other interesting features to include are inline method and classes implementations, and multithreading programming; to mention a few. Probably each of those features might require an individual research work, but some of the related work materials provide a good starting point.

There are some scenarios where we could have supported controlled cycles, because there are cyclic data structures used in the real world that can be turned into acyclic (or acyclic enough) without efficiency loss. Doubly-Linked lists might be a good starting point to work on. An interesting code to show would be the Windows Device Driver presented in [LQG+09].

Singularity Project[HLA+05] presented an experimental work on how to create an operating system from scratch, using contracts and advanced programming techniques to accomplish a more reliable system. That experience can lead to an interesting path of future work to create a memory management system that understands the acyclicity property of memory objects, and therefore use a cheaper garbage collection

algorithm. Minix[Tan10] should be enough to create a proof-of-concept. The results on the performance of the memory management system will serve to complete our study of acyclic objects.

# Bibliography


[App09] Apple. Memory management programming guide for Cocoa. http://developer.apple.com/iphone/, 2009.

[AS96] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 1996.

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[BCD+05] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[BL05] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *PASTE '05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 82–87, New York, NY, USA, 2005. ACM.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview, 2004.

[Boy03] John Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, Berlin, Heidelberg, New York, 2003. Springer.

[Bro75] Fred P. Brooks. The mythical man-month. In *Proceedings of the international conference on Reliable software*, page 193, New York, NY, USA, 1975. ACM.

[CKP+08] Silviu S. Craciunas, Christoph M. Kirsch, Hannes Payer, Ana Sokolova, Horst Stadler, and Robert Staudinger. A compacting real-time memory management system. In *ATC'08: USENIX 2008 Annual Technical Conference*, pages 349–362, Berkeley, CA, USA, 2008. USENIX Association.

[CPN98] David Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. *SIGPLAN Notes*, 33(10):48–64, 1998.

[Dij97] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin, April 2008.

[DVE00] Sophia Drossopoulou, Tanya Valkevych, and Susan Eisenbach. Java type soundness revisited, 2000.

[Ern08] Michael D. Ernst. Type Annotations specification (JSR 308). http://types.cs.washington.edu/jsr308/, 2008.

[FFA99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 192–203, New York, NY, USA, 1999. ACM.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proceedings of a Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–31, Providence, 1967. American Mathematical Society.

[FS01] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. *SIGPLAN Not.*, 36(3):193–205, 2001.

[HK00] Kees Huizing and Ruurd Kuiper. Verification of object oriented programs using class invariants. In *FASE '00: Proceedings of the Third International Conference on Fundamental Approaches to Software Engineering*, pages 208–221, London, UK, 2000. Springer-Verlag.

[HLA+05] Galen Hunt, James R. Larus, Martin Abadi, Mark Aiken, Paul Barham, Manuel Fahndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian D. Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, October 2005.

[Hoa83] Tony Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 26(1):53–56, 1983.

[HWG03] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Kas06] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2006.

[Kas08] Ioannis T. Kassios. A theory of object oriented refinement. PhD thesis, University of Toronto, 2008.

[Koc03] Stephen Kochan. *Programming in Objective-C*. Sams, Indianapolis, IN, USA, 2003.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design, 1999.

[Lei08a] K. Rustan M. Leino. Specification and verification of object-oriented software. Marktoberdorf International Summer School, lecture notes, 2008.

[Lei08b] K. Rustan M. Leino. This is Boogie 2. Manuscript KRML 178, 2008.

[Lei10] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. Manuscript KRML 203, 2010.

[LG86] Barbara Liskov and John Guttag. *Abstraction and specification in program development*. MIT Press, Cambridge, MA, USA, 1986.

[LLM07] Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.

[LM09] K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, pages 378–393, Berlin, Heidelberg, 2009. Springer-Verlag.

[LMS09] K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *FOSAD*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer, 2009.

[LP04] Yi. Lu and John Potter. *On Reachability and Acyclicity.* University of New South Wales, School of Computer Science and Engineering, 2004.

[LQG+09] Shuvendu K. Lahiri, Shaz Qadeer, Juan P. Galeotti, Jan W. Voung, and Thomas Wies. Intra-module inference. In *CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification*, pages 493–508, Berlin, Heidelberg, 2009. Springer-Verlag.

[Mey91] Bertrand Meyer. *Design by Contract, in Advances in Object-Oriented Software Engineering.* Prentice Hall, 1991.

[Mey92] Bertrand Meyer. *Eiffel: the language.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[Mic09a] Microsoft. Code contracts. http://research.microsoft.com/en-us/projects/contracts/, 2009.

[Mic09b] Microsoft. Microsoft Research Boogie. http://boogie.codeplex.com/, 2009.

[Par72] David L. Parnas. A technique for software module specification with examples. *Commun. ACM*, 15(5):330–336, 1972.

[PB08] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 75–86, New York, NY, USA, 2008. ACM.

[PHM99] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In *ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems*, pages 162–176, London, UK, 1999. Springer-Verlag.

[Pie02] Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.

[Ser97] Manuel Serrano. Inline expansion: When and how? In *PLILP '97: Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 143–157, London, UK, 1997. Springer-Verlag.

[SJP08] Jan Smans, Bart Jacobs, and Frank Piessens. Vericool: An automatic verifier for a concurrent object-oriented language. In *FMOODS '08: Proceedings of the 10th IFIP WG 6.1 international conference on Formal Methods for Open Object-Based Distributed Systems*, pages 220–239, Berlin, Heidelberg, 2008. Springer-Verlag.

[SJP09] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *Genoa: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, pages 148–172, Berlin, Heidelberg, 2009. Springer-Verlag.

[SJPS08] Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for Java-like programs based on dynamic frames. In José Luiz Fiadeiro and Paola Inverardi, editors, *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 261–275. Springer, 2008.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.

[Tan10] Andrew Tanenbaum. Minix. http://www.minix3.org/, 2010.

[Ter04] Pat D. Terry. *Compiling with C# and Java.* Addison Wesley, October 2004.