A proof system for object oriented programming using separation logic

Middelkoop, R.

Award date:
2003

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

MASTER'S THESIS

A Proof System for Object Oriented Programming using Separation Logic

by
Ronald Middelkoop

Supervisors: Dr. C. Huizing, Dr. R. Kuiper
Eindhoven, November 2003

Contents

Preface

The document before you is the master's thesis of Ronald Middelkoop. It was written in partial fulfilment of the requirements for graduation as a computing scientist at the Eindhoven University of Technology. This graduation project was done in the area of Software Technology for the research group Formal Methods.

The author would like to thank dr. E. J. Luit for his extensive proofreading and for pointing out the work of Poetzsch-Heffter and Muller.
He would also like to thank dr. Ruurd Kuiper and dr. Kees Huizing for their supervision over the project, for many hours of fruitful discussions, and for the occasional game of football.

Summary

As the complexity of and the responsibilities placed on programs increases, the need for effective automatic verification of programs based on their specification grows. One basic prerequisite for the development of such computer based tools is a verification formalism.

This project aims to develop a formalism for reasoning about object oriented programs. To this end, a small language is formulated in wich simple, yet interesting OO programs can be written. A complete logical framework surrounding this language is constructed from the ground up. This framework includes an operational semantics for the language, an assertion language that can express relevant properties of a program state and specifications that can express relations between pre- and postconditions of statements.

Central is a form of local reasoning called separation logic recently developed by Reynolds and O'Hearn. This local reasoning allows proofs to isolate those parts of an assertion a statement actually acts upon, automatically leaving the rest unchanged.

Inference rules, both structural and statement-specific are presented to enable the annotation and verification of programs in the language. Soundness for the statement-specific rules is proved.

A special interest was taken in the development of a simple, yet effective rule for local reasoning about method specifications and the subsequent calling of those methods. The applications of these rules are illustrated by several examples.

1 Introduction

This thesis presents the results of its authors graduation project. The project attempts to create a proof system for an object-oriented language by utilizing the concepts of separation logic, which allows for local reasoning about statements and methods.

This first chapter introduces the subject matter and present the foundation for the work that was done. It also formulates the goal of the project.

1.1 Background

Nowadays, software is found in almost any electrical device we use. As such, the tasks given to programs have diversified and increased in complexity. To accommodate for this increasing complexity, program development and program testing methodologies have been developed and are widely used. Verification methodologies however, that can prove correct a program with regards to a specification given for it, see very little use in everyday programming. This despite the fact that the advantages of verification over testing should theoretically increase with the complexity and size of the program. After all, testing, based on operational reasoning, has to account for *all* computations that can possibly be executed. In proving a program correct one can do without operational reasoning, greatly reducing the workload in the case of programs using repetition.

The opportunity to do away with operational reasoning is also the basis for another advantage that can reduce the complexity. The use of specifications allows one to speak in terms of *what* a (part of a) program should do, instead of *how* a program should perform such a task. The ability to group statements and describing the result of applying them to a given situation allows one to see them as a single program step. After all, it is not important how a program state is reached, but that it is reached.

But the most important advantage is the fact that, while testing might be a good way to find errors in the program code, it cannot give the certainty that a program behaves as is expected. This important advantage of verification could prove crucial as people place more and more responsibilities on programs.

However, to convert these advantages from theory to practice, a clear and simple formalism that is powerful enough to express the properties of interest to the program developer is required. Furthermore, instead of placing the burden of proof on a developer, these properties should be verified by an automated proof system. Unfortunately, on neither of these requirements a generally accepted solution has been developed. So, despite some thirty-odd years of research on the subject, very few programs are proven correct.

1.2 Separation Logic

One of the problem areas of modern day program verification is that of aliasing. Aliasing takes place when a single program resource is referenced from several distinct points. These problems do not lie with the proper operational axiomatization of the operations on such resources. Instead, the main difficulty lies in the mismatch between the simple intuitions about such operations and the complexity of their axiomatic treatment. For instance, an assignment to a shared resource is operationally very simple, but can affect the value of several syntactically unrelated expressions.

Quite recently a new approach to deal with such shared resources, based on early work by Burstall [2], was discovered independently by Reynolds [7] and O'Hearn and Ishtiaq [4]. Central to these approaches is a new logical connector that allows one to localize the effects of operations on shared resources.
A most recent overview of this logic, called separation logic, can be found in [8].

1.3 Object Oriented Languages

So far, this logic has been applied mostly to low-level imperative programs, where the shared resources are the individual heap cells.
This graduation projects aims to incorporate the concepts of separation logic in a higher order language based on the principles of Object Orientation.

Object Orientation is one of the design methodologies developed to deal with ever larger and more complex programs. It allows one to reason on a high level about the various tasks and subtasks that a program should perform. Furthermore, it has the additional advantage of the possibility for code reuse. This is accomplished by using objects as the main building blocks. Objects can be individually understood and then combined, integrated and arranged in different configurations. One main difference between the work done so far and the work done here is that objects will have to replace individual heap cells as the shared resources. One additional issue that has to be dealt with is that the level of the smallest shared resource, the object field, is not equal to the level on which sharing takes place, the object level.

1.4 Goal

The goal of this project is to create clear and simple inference rules that allow one to annotate and prove correct programs written in an object oriented language. When design choices are made, it is always with this goal in mind. Problems that do not have a direct relation with the problem of shared resources are to be ignored or abstracted from wherever possible if they would otherwise complicate the inference rules.

2 Language Grammar

The proof system central to this document can prove properties of programs written in the language presented in this chapter. This language was developed to accommodate the concepts needed. It aims to be as simple as possible while still offering the possibility of writing object-oriented programs of practical value. It is based on the language fragment from [4]. The presentation is based on [1].

This document assumes the existence of several sets. For each of these sets a so called *typical element* is given. A typical element, and variations on the typical element, are used as meta-variables ranging over their set. Variations include priming and indexing of the typical element.

Listed below are the names, contents and typical elements of the sets assumed to exist.

Name	**Contains**	**Typical Element**
$IntLit$	$\mathbb{Z}$	i
$BoolLit$	$\{\textbf{true}, \textbf{false}\}$	b
$ClassName$	class names	c
$MethodName$	method names	m
$InstVar$	instance variables	a
$TempVar$	temporary variables	x
$Type$	$\{Int, Bool, Obj\}$	θ
$DataType$	$\{Int, Obj\}$	t

As can be seen, the set $IntLit$ is the set of *integer literals*. This set consists of all integer values, for example 0,1, -14 and 23.

In order not to get distracted by the separate concern of static type correctness we use typed expressions and decorate the syntax with relevant type information. The set $Type$ contains all types used. This includes the two primitive types Int en $Bool$. Subtyping is considered as yet another separate concern. Although it is not really dealt with in this thesis, a very basic form is in place. The type Obj is the only reference data type, and it is used implicitly as a supertype for all classes.

Naturally, the elements of the sets of literals have their respective types. As a further simplification, only Int and Obj are considered as possible types for variables. For that purpose, t is used as the typical element of the set of data types $\{Int, Obj\}$. It is because of this simplification that no other sets of literals besides those given are needed.

The set $InstVar$ consists of all possible *instance variables*, also known as *fields*, with typical element a. This set is constructed from the union of the two sets $InstVar^t$, the sets of instance variables of type t, with typical element a^t.
The set $TempVar$ consists of all possible *temporary variables*, also known as *local variables*, with typical element x. It too is constructed from two sets, known as $TempVar^t$, the sets of temporary variables with type t, with typical element x^t.

Now the syntax of the language can be specified.

$ObjectRef$ consists of those variables that might refer to an object without using ., the dereferencing operator or dot-operator. Typical element of the set is o:

$$o \ ::= \ x^{Obj} \mid \textbf{this}$$

As is usual in OO-notation, **this** is a special keyword variable used to reference the object of which a method is being executed from within that method. Note that o's type is Obj by definition, preventing the need to mention its type explicitly.

Define the set Exp, typical element e, as the set of expressions in the language. This set is constructed from the union of the three sets Exp^{Int}, Exp^{Obj} and Exp^{Bool}, with typical elements e^{Int}, e^{Obj} and e^{Bool} respectively. The set Exp^{Int}, the set of integer expressions, is defined as:

$$e^{Int} \quad ::= \quad i \mid x^{Int} \mid e_1^{Int} + e_2^{Int} \mid e_1^{Int} - e_2^{Int} \mid e_1^{Int} \times e_2^{Int}$$

The set of object expressions Exp^{Obj}, is defined as:

$$e^{Obj} \quad ::= \quad o \mid \textbf{nil}$$

The special keyword constant **nil** is used to signal that a variable does not refer to an object. For reasons given in the syntax of statements, Exp^{Obj} does not contain elements containing the dot-operator (and consequently no elements containing instance variables).

The third and last set used in the construction of Exp is the set Exp^{Bool}, the set of boolean expressions. It is defined as:

$$e^{Bool} \quad ::= \quad b \mid e_1 = e_2 \mid e_1^{Int} > e_2^{Int} \mid e_1^{Bool} \wedge e_2^{Bool} \mid e_1^{Bool} \vee e_2^{Bool} \mid \neg e^{Bool}$$

Using the sets defined so far the set $Stat$ of statements in the language can now be specified.
The formal operational meaning of these statements is given in chapter 4.
This set, with typical element S, is described in several parts, where each part of the definition is accompanied by some explanation. The first part describes the assignment statement. As can be seen in chapter 10, in order to write clear inference rules for assignment it is important that an assignment statement contains no more than one dot-operator. To this end the classical assignment statement is split up, and several cases are excluded. Among these are assignments to the **this** variable.

$$\begin{array}{lll} S & ::= & x^t := e^t \\ & \mid & x^t := o.a^t \\ & \mid & o.a^t := e^t \end{array}$$

Note that this does not constitute a true limitation of the syntactic power of the language. By using fresh auxiliary variables, an assignment with more than one dereferencing can always be rewritten to a series of equivalent assignments that use no more than one dereferencing each. Also note that type correctness is syntactically enforced.

There is one other form of assignment, used to create a new object. This form uses the keyword **new**.

$$\mid \quad x^{Obj} := \textbf{new}\ c$$

In this version of the language, methods do not have parameters and do not return a value. Therefore, a method call is simply written as:

$$\mid \quad o.m$$

Parameters and return values are considered as an extension to the language in chapter 12.

One of the interesting aspects of this logic is that explicit object deallocation is possible. This can be done using the following statement:

$$\mid \quad \textbf{dispose}(o)$$

Note that even the statement **dispose(this)** is syntactically possible.

The statement **skip** is intended not to alter the program state:

$$\mid \quad \textbf{skip}$$

Composite statements are constructed using the familiar methods of sequential composition and choice:

$$\mid \quad S_1; S_2$$
$$\mid \quad \textbf{if } e^{Bool} \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}$$

The while-statement, used for iteration, is not in the language for reasons of simplicity and to force the use of method call, the most interesting statement from a proof-theoretic point of view.

MethodDeclaration, with typical element μ, also shows that methods do not have parameters or a return value. A method declaration therefore consists of the method's name m and the statement that constitutes its body S:

$$\mu \quad ::= \textbf{method } m \ \{S\}$$

Classes serve as the blueprint for the objects instantiated from them using the **new** assignment. Class declarations are members of the set *ClassDeclaration*, with typical element C. A class is identified by it's name c, and specifies zero or more fields a_i and methods μ_i:

$$C \quad ::= \textbf{class } c \ \{a_1, \ldots, a_n; \ \mu_1; \ldots; \ \mu_m\}$$

Finally, *ProgramDeclaration*, typical element *PDecl*, is the set of programs in the grammar. A program consists of zero or more class declarations and a **main** statement:

$$PDecl \quad ::= C_1; \ldots; \ C_n; \ \textbf{main } \{S\}$$

The set *ProgramDeclaration* contains certain program declarations with undesirable properties, that cannot be easily expressed in a context free grammar. Therefore, the set *ProgramDeclaration'*, typical element *PDecl'*, is the subset of *ProgramDeclaration* of program declarations that are *statically correct*, where an element *PDecl* is statically correct if and only if

- *PDecl* is finite
- all classes C_i of *PDecl* have a different name c;
- All fields a_i of such a C_i are different, and all methods μ_i of C_i have a different name m
- for every occurrence of a statement $x^{Obj} :=$ **new** c in such a method μ_i's body S or in *PDecl*'s **main** statement, c is the name of a class C_i in *PDecl*
- To prevent problems stemming from uninitialized variables (note that variables cannot be declared), every temporary variable occurring on the right-hand side of an assignment in a statement S must occur earlier on the left-hand side of an assignment in that same statement S

Now a program in the language is an element of the set *ProgramDeclaration'*. Such a program starts its execution with the statement in **main**.

3 Operational Model

In the construction of the language grammar, a choice was made to make the elements of the set $IntLit$ the only data values allowed. This way there is little distraction from data-related problems. As a result of this choice, the set $IntLit$ is sufficient as a data domain for the model. Besides data, there is one other important category of model values, this being *locations*. Locations are used as unique object identifiers.
Unlike in the models of O'Hearn [4] and Reynolds [8], where locations, like data, are modelled by integers, a separate set is introduced for these locations. As the possibilities of using one set (most specifically address arithmetic) are not very relevant in an OO-setting, a better separation of concerns is achieved at little cost.

The set $\mathcal{A}$, typical element α, is an infinite set of locations.

The set $\mathcal{D}$ has δ as its typical element and consists of all possible model values. As stated above, integers are the only data values. This means that this set is the union of the set of locations, the set of integer values and the singleton set with **nil** as its only element. Note that **nil** is used when a reference variable does not refer to an object.

$$\mathcal{D} \quad = \quad IntLit \cup \mathcal{A} \cup \{\mathbf{nil}\}$$

The operational state has two components, a $Store$ and a $FieldHeap$. The Store is a function mapping a temporary variable or the variable **this** to a model value.

$$Store \quad : \quad TempVar \cup \{\mathbf{this}\} \rightarrow_{par} \mathcal{D}$$

Typical element of this function is s.
$\rightarrow_{par}$ is used for functions that are both finite and partial.
A constraint on this function is that the variable **this** may only be mapped to an element of $\mathcal{A}$. Furthermore, elements x^{Int} must be mapped to $IntVal$ and elements x^{Obj} must be mapped to $\mathcal{A}$. When s maps two variables to the same value in $\mathcal{A}$ these variables are known as *aliases*.

The $FieldHeap$ is used to provide each object with one unique location. An object is characterized by its class, which determines the methods the object can execute, and by the values of its instance variables, which can be identified by their names. As can be seen later on in chapters 7 and 11, the exact definition of this function is of great importance in this logic. The $FieldHeap$ is defined as:

$$FieldHeap \quad : \quad \mathcal{A} \rightarrow_{par} ClassName \times (InstVar \rightarrow_{par} \mathcal{D})$$

As a typical element of this function f is used.
Because every object is identified by exactly one location the first function application is required to be injective.
With $f(\alpha)\rfloor_i$, $i \in 1,2$ the first or second projection on the result of applying f to α is meant.
Note that situations where a single field has more than one name are not considered, but that the lowest level of aliasing is the object level.
To improve readability $f(\alpha, a)$ is written instead of $f(\alpha)\rfloor_2(a)$ where possible. Note that this can be done unambiguously, as only the second projection takes another argument.

Now when a variable x^{Obj} refers to an object in a certain state, this can be expressed by having the $Store$ function map x^{Obj} to a location α. This α is then mapped by the $FieldHeap$ function to the class of the object and a set of names of the instance variables defined for that object, which in turn are each mapped to the value of the variable in that particular state.

Taking an $s \in Store$ and an $f \in FieldHeap$ the tuple $Opstate = Store \times FieldHeap$ represents the

operational state. Using *OpState* it is possible to define *configurations*, which come in 2 flavors:

- A *terminal* configuration is a state $(s, f) \in Opstate$ or **abort**
- A *non-terminal* configuration is a statement-state pair $< S, (s, f) >$

Here, **abort** is a special configuration used to signal a reference fault, for example the dereferencing of a pointer that does not refer to an object.

Note that a statement S is always part of a program declaration $PDecl'$. This $PDecl'$ is an implicit, static, part of the configuration, and it is assumed that syntactical information can be extracted from it.

It is on these configurations that the operational semantics are defined.

4 Operational Semantics

Using the model described in the previous chapter it is possible to give an operational meaning to the statements in the set *Stat*. This is done by defining a natural semantics for these statements.

There are several advantages to specifying an operational semantics. First of all, this captures the functionality of the statements in a formal way, a goal all by itself. Furthermore, the formal way of capturing this functionality comes fairly close to the intuitive image one has of the functionality, or in other words, the statements do what you expect them to do. Additionally, no novel operators are necessary for the specification. Because of these points, the discussion on the correctness of the semantics boils down to mostly notational aspects. This makes the operational semantics a good basis for when we later want to show the correctness of the novel aspects and operators of the logic by means of soundness proofs.

The operational semantics focusses on only one sort of operational fault, called reference fault. For this category of faults the transitions lead to the terminal configuration **abort**. The proof system guarantees that faults in this category do not occur in a well specified program, someting that is crucial to the scalability of such specifications. Other categories of faults, like infinite recursion, cause a configuration to become stuck, and yet others have been excluded syntactically. This category includes most faults generated by the use of uninitialized temporary variables. Note that the prevention of such faults in well specified programs is certainly technically feasible, but leads to a more complex proof system, distracting from and obscuring the central problem this logic tries to deal with (i.e. dealing with shared mutable resources in an OO setting). When this problem is solved to satisfaction, extensions of this logic can take these other categories into account.

Natural semantics are used instead of small step or structured operational semantics as are used by Reynolds [8] and O'Hearn [9]. The main reason for this is the operational interpretation of the method call statement. Natural semantics allow one to view the execution of a method call as a single step, making for a simpler and clearer rule that can directly relate the state after to the state before the call. The price paid for this is a larger number of rules for the statements for sequential composition and choice. A semantics that is a mixture of natural and small step semantics was considered, but not used here to prevent discussion about what is to be used as the basis of the proof system. The natural semantics for the method call are based for a large part on those given in [6].

The semantics define a partial transition relation $\rightsquigarrow$ between configurations.
$initS \in Store$ is the *Store*-function with empty domain.
$dom(g)$ gives the domain of function g, $g\restriction D$ denotes the restriction of g to the domain D. $(g \mid M)$ is the function like g, but with the mappings as described by M, where M is defined by:

$$M \quad ::= \quad k \rightarrow l \mid M_1, M_2$$

Here, $k \rightarrow l$ means element k is mapped to l, or in other words that $g(k) = l$. M_1, M_2 is the simultaneous application of mappings M_1 and M_2. Note that k is not required to be in $dom(g)$ for this notation to be used.
As an example, when function g is defined by $g(1) = 1$, the function $(g \mid 1 \rightarrow 2, 3 \rightarrow 4)$ is defined by $g(1) = 2$ and $g(3) = 4$.

Also needed is a valuation that maps expressions to $\mathcal{D}$ using the *Store* function:

$$[\![e^t]\!] : Store \rightarrow_{par} \mathcal{D}$$

Its semantics are:

$$
\begin{array}{lcl}
||[x^t]||_s & = & s(x^t) \\
||[\textbf{this}]||_s & = & s(\textbf{this}) \\
||[\textbf{nil}]||_s & = & \textbf{nil} \\
||[i]||_s & = & i
\end{array}
\qquad
\begin{array}{lcl}
||[e_1^{Int} + e_2^{Int}]||_s & = & ||[e_1^{Int}]||_s + ||[e_2^{Int}]||_s \\
||[e_1^{Int} - e_2^{Int}]||_s & = & ||[e_1^{Int}]||_s - ||[e_2^{Int}]||_s \\
||[e_1^{Int} \times e_2^{Int}]||_s & = & ||[e_1^{Int}]||_s \times ||[e_2^{Int}]||_s
\end{array}
$$

A second valuation maps boolean expressions to their truth value:

$$||[e^{Bool}]|| : Store \rightarrow_{par} BoolLit$$

It has the following (fairly obvious) semantics:

$$
\begin{array}{lcl}
||[b]||_s & = & b \\
||[e_1 = e_2]||_s & = & \text{if } ||[e_1]||_s = ||[e_2]||_s \text{ then } \textbf{true} \text{ else } \textbf{false} \\
||[e_1^{Int} > e_2^{Int}]||_s & = & \text{if } ||[e_1^{Int}]||_s > ||[e_2^{Int}]||_s \text{ then } \textbf{true} \text{ else } \textbf{false} \\
||[e_1^{Bool} \wedge e_2^{Bool}]||_s & = & \text{if } ||[e_1^{Bool}]||_s = \textbf{true} \text{ and } ||[e_2^{Bool}]||_s = \textbf{true} \text{ then } \textbf{true} \text{ else } \textbf{false} \\
||[e_1^{Bool} \vee e_2^{Bool}]||_s & = & \text{if } ||[e_1^{Bool}]||_s = \textbf{true} \text{ or } ||[e_2^{Bool}]||_s = \textbf{true} \text{ then } \textbf{true} \text{ else } \textbf{false} \\
||[\neg e^{Bool}]|| & = & \text{if } ||[e_1^{Bool}]||_s = \textbf{true} \text{ then } \textbf{false} \text{ else } \textbf{true}
\end{array}
$$

Let the relation $\rightsquigarrow$ be the smallest relation that satisfies the following set of transition rules:

1. The first transition rule describes the meaning of the statement $x^t := e^t$. After execution of this statement variable x^t has the value of expression e^t. Because this expression is by definition independent of the $FieldHeap$ function f no reference fault can occur. Note however that this configuration would get stuck when e^t cannot be evaluated, a situation that should not occur because of the syntactical constraints on programs and the other transition rules.

$$\frac{||[e^t]||_s = \delta}{< x^t := e^t, (s, f) > \rightsquigarrow ((s \mid x^t \rightarrow \delta), f)}$$

2. The statement $x^t := o.a^t$ is expected to map x^t to the value of the field a^t of the object referred to by o. If o does not refer to an object with a field a^t this operation results in a reference fault.

$$\frac{o \in dom(s) \quad s(o) \in dom(f) \quad a^t \in dom(f(s(o)))}{< x^t := o.a^t, (s, f) > \rightsquigarrow ((s \mid x^t \rightarrow f(s(o), a^t)), f)}$$

$$\frac{o \notin dom(s) \text{ or } s(o) \notin dom(f) \text{ or } a^t \notin dom(f(s(o)))}{< x^t := o.a^t, (s, f) > \rightsquigarrow \textbf{abort}}$$

   Note that $o \notin dom(s)$ is technically not a reference fault (but an initialization fault). However, stating it here explicitly makes for the most unambiguous situation without adding complexity to the proof system.

3. $o.a^t := e^t$ assigns the value that is represented by expression e^t to instance variable a^t of the object referred to by o. Again, if o does not refer to an object with a field a^t this operation results in a reference fault.

$$\frac{||[e^t]||_s = \delta \quad o \in dom(s) \quad s(o) \in dom(f) \quad a^t \in dom(f(s(o)))}{< o.a^t := e^t, (s, f) > \rightsquigarrow (s, (f \mid (s(o), a^t) \rightarrow \delta))}$$

$$\frac{o \notin dom(s) \text{ or } s(o) \notin dom(f) \text{ or } a^t \notin dom(f(s(o)))}{< o.a^t := e^t, (s, f) > \rightsquigarrow \textbf{abort}}$$

4. The statement $x^{Obj} := \textbf{new } c$ creates a new object, which is an instance of class c. After execution variable x^{Obj} refers to the new object.

Because an object is represented in the $FieldHeap$ function by its instance variables, access to the names of these variables is needed. Therefore, given a $c \in ClassName$, the function ClassFields gives the set of instance variables as defined by c's class definition. Recall that only programs where every occurrence of this statement refers to a c that is defined are considered (see chapter 2):

$$ClassFields \quad : \quad ClassName \rightarrow \text{Set of } InstVar$$

In order to prevent having to make an explicit choice for the initial values of the instance variables of the object, the function $InitVal$ is introduced:

$$InitVal \quad : \quad ClassName \times InstVar \rightarrow_{par} \mathcal{D}$$

This function gives an initial value to every valid tuple of a class name and an instance variable. A tuple is valid when the class definition of the class name defines the instance variable, i.e. a tuple (c, a) is valid if and only if $a \in ClassFields(c)$.

These two functions are used to create the total functions $g_c : ClassFields(c) \rightarrow \mathcal{D}$, defined by $g_c(a) = InitVal(c, a)$. When the $FieldHeap$ function is updated, such a function application can serve as the second element of the tuple representing the new object.

Last but not least a fresh location (i.e a location that does not currently identify an object) is needed to serve as the unique identifier for the new object. It is assumed that such a location is always available (note that the set of locations is infinitely large). The choice of the location is non-deterministic. Why this is essential is described in [9]. Now we can define:

$$\frac{\alpha \in (\mathcal{A} - dom(f))}{< x^{Obj} := \textbf{new } c, (s, f) > \rightsquigarrow \ ((s \mid x^{Obj} \rightarrow \alpha), (f \mid \alpha \rightarrow (c, g_c)))}$$

5. The most complex operation is that of the method call.
A method call $o.m$ is expected to execute the body of the method specified. This operation only succeeds if o does indeed refer to an object, and if the object is an instance of a class that has a method m.

   Besides these requirements, $o.m$ can only transition to a non-**abort** state when the body of the method can be executed safely. However, this body cannot use the temporary variables of the current state, because these variables are local to the calling method. Therefore, execution must be possible in a state in which the only local variable is the special variable **this**, which gives the called method access to the object the method is executed on (i.e the object referred to by o). The scope of this $Store$ function is the body of the method, meaning that it is discarded when the body terminates. At that point the original $Store$ function is restored.

   So, the transition rule's terminal configuration has an unchanged $Store$ function because none of the variables it maps can be changed by the body, while the $FieldHeap$ function is updated to reflect the possible changes made by the body of method m.

   Checking if the object referred to by o can execute a method m is done using:

$$Methods \quad : \quad ClassName \rightarrow \text{Set of } MethodName$$

   To access the statement that constitutes the body the function $Body$ is used:

$$Body \quad : \quad ClassName \times MethodName \rightarrow Stat$$

This makes for the following transition rule:

$$\frac{o \in dom(s) \quad s(o) \in dom(f) \quad m \in Methods(f(s(o))\rfloor_1) \quad < Body(f(s(o))\rfloor_1, m), ((initS \mid \textbf{this} \rightarrow s(o)), f) > \rightsquigarrow \ (s', f')}{< o.m, (s, f) > \rightsquigarrow \ (s, f')}$$

If one of the aforementioned requirements is not met, a reference fault occurs. The transition leads to **abort**:

$$\frac{o \notin dom(s) \text{ or } s(o) \notin dom(f) \text{ or } m \notin Methods(f(s(o))\rfloor_1) \text{ or } < Body(f(s(o))\rfloor_1, m), ((initS \mid \textbf{this} \rightarrow s(o)), f) > \rightsquigarrow \ \textbf{abort}}{< o.m, (s, f) > \rightsquigarrow \ \textbf{abort}}$$

Note that infinite recursion in the body of the method causes this configuration to get stuck.

6. **dispose**(o) simply removes the object referred to by o from the $FieldHeap$ function. Note that this makes o a *dangling pointer*, i.e a variable with a value that is a location that is not mapped to an object.

$$\frac{o \in dom(s) \quad s(o) \in dom(f)}{< \textbf{dispose}(o), (s, f) > \rightsquigarrow \ (s, (f \restriction (dom(f) - \{s(o)\})))} \qquad \frac{o \notin dom(s) \text{ or } s(o) \notin dom(f)}{< \textbf{dispose}(o), (s, f) > \rightsquigarrow \ \textbf{abort}}$$

7. The statement **skip** is designed not to make any changes to the state, making its semantics extremely simple:

$$\frac{}{< \textbf{skip}, (s, f) > \rightsquigarrow \ (s, f)}$$

8. The transition for sequential composition decomposes the two statements and makes execution of the second dependent on the successful and safe execution of the first:

$$\frac{< S_1, (s_1, f_1) > \rightsquigarrow \ (s_2, f_2) \quad < S_2, (s_2, f_2) > \rightsquigarrow \ (s_3, f_3)}{< S_1; S_2, (s_1, f_1) > \rightsquigarrow \ (s_3, f_3)}$$

$$\frac{< S_1, (s, f) > \rightsquigarrow \ \textbf{abort}}{< S_1; S_2, (s, f) > \rightsquigarrow \ \textbf{abort}} \qquad \frac{< S_1, (s_1, f_1) > \rightsquigarrow \ (s_2, f_2) \quad < S_2, (s_2, f_2) > \rightsquigarrow \ \textbf{abort}}{< S_1; S_2, (s_1, f_1) > \rightsquigarrow \ \textbf{abort}}$$

9. The if statement executes one of its two branches based on the truth value of its guard:

$$\frac{|[e^{Bool}]|_s = \textbf{true} \quad < S_1, (s, f) > \rightsquigarrow \ (s', f')}{< \textbf{if } e^{Bool} \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, (s, f) > \rightsquigarrow \ (s', f')}$$

$$\frac{|[e^{Bool}]|_s = \textbf{true} \quad < S_1, (s, f) > \rightsquigarrow \ \textbf{abort}}{< \textbf{if } e^{Bool} \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, (s, f) > \rightsquigarrow \ \textbf{abort}}$$

$$\frac{|[e^{Bool}]|_s = \textbf{false} \quad < S_2, (s, f) > \rightsquigarrow \ (s', f')}{< \textbf{if } e^{Bool} \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, (s, f) > \rightsquigarrow \ (s', f')}$$

$$\frac{|[e^{Bool}]|_s = \textbf{false} \quad < S_1, (s, f) > \rightsquigarrow \ \textbf{abort}}{< \textbf{if } e^{Bool} \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, (s, f) > \rightsquigarrow \ \textbf{abort}}$$

Note that the configuration becomes stuck if e^{Bool} cannot be evaluated.

5 Assertion Language Grammar

Assertions are used to describe certain properties of a state. The assertion language grammar given in this chapter is a first basic attempt to describe relevant characteristics, aimed at capturing first order properties of the programming language. Chapter 12 expands this language with more complex properties.

The assertion syntax assumes two additional sets as given. The first is the set $ExiVar$, the set of existentially quantified (logical) variables. Typical element for this set is d. The other is $UniVar$, the set of universally quantified logical variables, with typical element D. Note that these variables cannot occur in the program and that their values cannot be changed by the program.

In the assertion language, the set of elements that can refer to an object without using the dereferencing operator is the extension of the set $ObjectRef$ with the sets $ExiVar$ and $UniVar$. This set, known as $AObjectRef$ has typical element O and is defined by:

$$O \quad ::= \quad o \mid d \mid D$$

Expressions are extended with the elements from the two sets as well. Analog to Exp, The set $AExp$, typical element E, is the union of the three sets $AExp^{Int}$, $AExp^{Obj}$ and $AExp^{Bool}$, which in turn are defined by:

$$E^{Int} \quad ::= \quad i \mid x^{Int} \mid d \mid D \mid E_1^{Int} + E_2^{Int} \mid E_1^{Int} - E_2^{Int} \mid E_1^{Int} \times E_2^{Int}$$

$$E^{Obj} \quad ::= \quad O \mid \mathbf{nil}$$

$$E^{Bool} \quad ::= \quad b \mid E_1 = E_2 \mid E_1^{Int} > E_2^{Int}$$

Note that not all of the elements specified as boolean expressions in the program language translate to the assertion language's set of boolean expressions. This is because the other expressions are defined on the level of propositions, specified below.

Objects take a central place as the mutable shared resources in this logic dedicated to dealing with some of the problems stemming from such resources. A special notation is introduced to describe objects.

An object is written as $O \mapsto [a_1 : E_1^t, \ldots, a_n : E_n^t]^c$, where $n \geq 0$ and $a_1, \ldots a_n$ are all different. It is not possible to describe a single field. More on the design decisions regarding this notation can be found in chapter 11.

Two other new forms of propositions also describe properties of the heap. emp is used to signal the heap is empty, and $*$ is the crucial operator that divides the shared resources into two distinct groups.

The language's set of propositions, $AProp$, with typical element P, is defined below.
The formal meaning of these propositions is given in chapter 7.

$$P \quad ::= \quad E^{Bool} \mid P_1 \Rightarrow P_2 \mid \exists d.\ P \mid O \mapsto [a_1 : E_1^t, \ldots, a_n : E_n^t]^c \mid emp \mid P_1 * P_2$$

Note that some other well-known propositions can be constructed from $AProp$:
$\neg P = P \Rightarrow false; true = \neg(false); P_1 \vee P_2 = (\neg P_1) \Rightarrow P_2; P_1 \wedge P_2 = \neg(\neg P_1 \vee \neg P_2); \forall d.P = \neg\exists d.\neg P$

Addionally, it is convenient to define the symbol $\doteq$, referred to as *exact equality*:

$$E_1 \doteq E_2 \quad = \quad (E_1 = E_2 \wedge emp)$$

6 Assertion Model

The assertion language defined in the previous chapter uses quantified elements not in the programming language. To interpret these elements, two functions are defined:

$$\begin{array}{lcl} Exi & : & ExiVar \rightarrow_{par} \mathcal{D} \\ Uni & : & UniVar \rightarrow_{par} \mathcal{D} \end{array}$$

Typical elements are q and u, respectively.

Analog to the *OpState* tuple that describes the operational state, the *AsState* tuple is defined as the collection of functions capturing all relevant dynamic information needed to interpret a proposition. However, as stated at the beginning of chapter 5, the language is enriched in chapter 12. Therefore, no exact definition of *AsState* is given here.
For now, the demand is simply that an *AsState* tuple contains functions s, f, q and u, because with these functions every element of the grammar defined so far can be interpreted. Both σ and τ are used as typical elements of the *AsState* tuple.

The projection of σ on the function g is written as g_σ.
$(\sigma \mid M)$ is the tuple like σ, but with the mapping M applied, where M is defined by:

$$M \quad ::= \quad g_1 \rightarrow g_2 \mid g(k \rightarrow l) \mid g(\forall j.\ 0 \leq j \leq i.\ k_j \rightarrow l_j) \mid M_1, M_2$$

Here, $g_1 \rightarrow g_2$ means function g_1 is replaced by function g_2, $g(k \rightarrow l)$ means element k is mapped to l in function g, $g(\forall j.\ 0 \leq j \leq i.\ k_j \rightarrow l_j)$ has g map elements k_j to l_j and M_1, M_2 is the simultaneous application of mappings M_1 and M_2.

As with the operational model (see chapter 3), it is assumed that static syntactical information can be extracted from the program declaration. The functions defined for this purpose in chapter 4 are therefore considered an implicit part of the model here as well.

The next chapter gives the interpretation of the assertion langauge grammar.

7 Semantics of Assertions

A proposition is either true or false in a given state. The semantics of assertions define when a proposition holds. To this end, the *satisfaction judgement* $\models$ is defined. When a proposition P is true in a state σ we write $\sigma \models P$.

The definition of $\models$ uses a valuation

$$|[E^t]| : Store \times ExiVar \times UniVar \rightarrow_{par} \mathcal{D}$$

This valuation maps the expressions of a proposition to elements of $\mathcal{D}$.
The valuation

$$|[E^{Bool}]| : Store \times ExiVar \times UniVar \rightarrow_{par} BoolLit$$

is used to do the same for boolean expressions, mapping them to **true** or **false**.
The semantics of these valuations are:

$$\begin{array}{llllllllll} |[x^t]|_{s,q,u} & = & s(x^t); & |[i]|_{s,q,u} & = & i; & |[E_1 + E_2]|_{s,q,u} & = & |[E_1]|_{s,q,u} + |[E_2]|_{s,q,u}; \\ |[\mathbf{this}]|_{s,q,u} & = & s(\mathbf{this}); & |[d_i]|_{s,q,u} & = & q(d_i); & |[E_1 - E_2]|_{s,q,u} & = & |[E_1]|_{s,q,u} - |[E_2]|_{s,q,u}; \\ |[\mathbf{nil}]|_{s,q,u} & = & \mathbf{nil}; & |[D_i]|_{s,q,u} & = & u(D_i); & |[E_1 \times E_2]|_{s,q,u} & = & |[E_1]|_{s,q,u} \times |[E_2]|_{s,q,u}; \end{array}$$

$$\begin{array}{lll} |[b]|_{s,q,u} & = & b \\ |[E_1 = E_2]|_{s,q,u} & = & \text{if } |[E_1]|_{s,q,u} = |[E_2]|_{s,q,u} \text{ then } \mathbf{true} \text{ else } \mathbf{false} \\ |[E_1^{Int} > E_2^{Int}]|_{s,q,u} & = & \text{if } |[E_1^{Int}]|_{s,q,u} > |[E_2^{Int}]|_{s,q,u} \text{ then } \mathbf{true} \text{ else } \mathbf{false} \end{array}$$

To prevent having to use a double subscript notation $|[E^t]|_\sigma$ and $|[E^{Bool}]|_\sigma$ are written instead of $|[E^t]|_{s_\sigma,q_\sigma,u_\sigma}$ and $|[E^{Bool}]|_{s_\sigma,q_\sigma,u_\sigma}$

The satisfaction judgement $\sigma \models P$ is defined by:

$$\begin{array}{lll} \sigma \models E^{Bool} & \text{iff} & |[E^{Bool}]|_\sigma = \mathbf{true} \\ \sigma \models P_1 \Rightarrow P_2 & \text{iff} & \text{if } \sigma \models P_1 \text{ then } \sigma \models P_2 \\ \sigma \models \exists d.\ P & \text{iff} & \exists \delta \in \mathcal{D} : (\sigma \mid q_\sigma(d \rightarrow \delta)) \models P \end{array}$$

So, $\exists d.\ P$ holds if and only if P holds in a state that is an extension of σ in such a way that q maps d to some δ.

$$\begin{array}{lll} \sigma \models O \mapsto [a_1 : E_1^t, \ldots, a_n : E_n^t]^c & \text{iff} & f_\sigma(|[O]|_\sigma)\rfloor_1 = c \text{ and} \\ & & (\forall i : 1 \leq i \leq n : f_\sigma(|[O]|_\sigma, a_i) = |[E_i^t]|_\sigma) \text{ and} \\ & & dom(f_\sigma) = \{|[O]|_\sigma\} \end{array}$$

As can be seen in this definition, three conditions must hold for $O \mapsto [a_1 : E_1^t, \ldots, a_n : E_n^t]^c$ to hold. The first two say that it holds just when O is a pointer to a location that represents an object of class c, with fields a_1 to a_n that have values equal to the values represented by E_1^t to E_n^t, respectively.

There is, however, a third condition that has to be met for the proposition to hold, which is referred to as *domain exactness*. The location that is the value of O must be the *only* location in the domain of the $FieldHeap$ function f_σ. This seemingly very restrictive demand actually makes the language stronger (as shown below) and allows for some very useful axioms (see 8.1 and 8.2). Additionally, this domain exactness, in conjunction with the logic's capability of dealing with dangling pointers, allows for the creation of an inference rule for the annotation of the **dispose** statement.

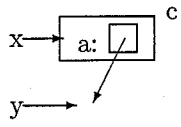
This domain exactness allows for a second exact statement. As can be seen below, *emp* asserts that the $FieldHeap$ function has an empty domain, or in other words that the $FieldHeap$ function does not define any objects.

$\sigma \models emp \quad \text{iff} \quad dom(f_\sigma) = \{\}$

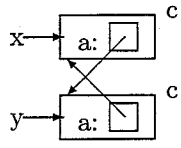
The final proposition is the logic's most innovative one. It is the new operator $*$, called *separating conjunction*. The separation conjunction gives a way to combine propositions as long as they don't rely on the same objects. In the definition below, $f_1 \# f_2$ means that the domains of functions f_1 and f_2 are disjoint. $f_1 \cdot f_2$ is the union of two such functions with disjoint domains.

$$\sigma \models P_1 * P_2 \quad \text{iff} \quad (\exists f_0, f_1 : f_0 \# f_1 \text{ and } f_\sigma = f_1 \cdot f_2 : (\sigma \mid f_\sigma \to f_1) \models P_1 \text{ and } (\sigma \mid f_\sigma \to f_2) \models P_2)$$

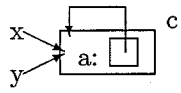
So, $P_1 * P_2$ holds if and only if it is possible to split the $FieldHeap$ function into two distinct parts, one making P_1 true and the other making P_2 true. This definition relies on the existence of dangling pointers in an essential way. For instance, $x \mapsto [a : y]^c$ (with $x, y \in TempVar^{Obj}$ and $a \in InstVar^{Obj}$) could be drawn as:



One could then extend the proposition using $*$ to $x \mapsto [a : y]^c * y \mapsto [a : x]^c$, which would be drawn as:

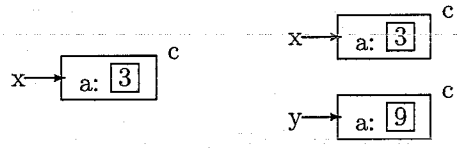


Here, it is the definition of $*$ that tells us x and y point to *different* objects as we know we can split the proposition into two parts, both having exactly one object. Likewise, the domain exactness makes us draw $x \mapsto [a : y]^c \wedge y \mapsto [a : x]^c$ as:



x and y must point to the same object as $x \mapsto [a : y]^c \wedge y \mapsto [a : x]^c$ is only true in a state where both $x \mapsto [a : y]^c$ and $y \mapsto [a : x]^c$ hold individually. Both of those propositions allow only a single object to be defined, which can only be the case when x and y are aliases.

Exact knowledge of the size of the domain of the $FieldHeap$ function, or in other words the number of objects defined by a proposition, is lost when $*$ is combined with propositions that are independent of the $FieldHeap$. For instance, in both situations drawn below $x \mapsto [a : 3]^c * \mathbf{true}$ (with $a \in InstVar^{Int}$) holds:



This way, the exact version of the logic can express the non-exact version using $*$, while the non-exact version is not able to express *emp*, making the first stronger than the second.

Some of the far-reaching consequences of using this type of operator can be seen in the next chapter, which gives axiom schemata for the logic.

8 Axiom Schemata

This chapter gives axiom schemata for the logic's new operators.

8.1 $*$ Axioms

The axioms given in this section are essentially those given in [9].

The separating conjunction is both associative and commutative. Its unit is *emp*, as the *FieldHeap* function with empty domain is disjoint from any other *FieldHeap* function:

$P_1 * P_2 \Leftrightarrow P_2 * P_1$
$(P_1 * P_2) * P_3 \Leftrightarrow P_1 * (P_2 * P_3)$
$P * emp \Leftrightarrow P$

$*$ distributes over $\vee$, and is semidistributive for $\wedge$.

$(P_1 \vee P_2) * P_3 \Leftrightarrow (P_1 * P_3) \vee (P_2 * P_3)$
$(P_1 \wedge P_2) * P_3 \Rightarrow (P_1 * P_3) \wedge (P_2 * P_3)$
$(\exists d.\ P_1) * P_2 \Leftrightarrow \exists d.\ (P_1 * P_2)$ when d not free in P_2
$(\forall d.\ P_1) * P_2 \Rightarrow \forall d.\ (P_1 * P_2)$ when d not free in P_2

Why $*$ is not fully distributive over $\wedge$ can be shown by the following example:
Let $P_1 = x \mapsto []^c$, $P_2 = y \mapsto []^d$ and $P_3 = (x \mapsto []^c \vee y \mapsto []^d)$. Then, as both $(P_1 * P_3)$ and $(P_2 * P_3)$ model the situation in which x refers to an object of type c and y refers to an object of type d, so would $(P_1 * P_3) \wedge (P_2 * P_3)$. $(P_1 \wedge P_2) * P_3$ however would read $(x \mapsto []^c \wedge y \mapsto []^d) * (x \mapsto []^c \vee y \mapsto []^d)$, which can not possibly hold.

Any proposition that is independent of the *FieldHeap* function is called *pure*. A proposition that does not contain a $\mapsto$ or *emp* is called *syntactically pure*. Any syntactically pure proposition is pure. When assertions are pure, the difference between the two types of conjunctions vanishes:

$P_1 \wedge P_2 \Rightarrow P_1 * P_2$ when P_1 or P_2 pure
$P_1 \wedge P_2 \Leftrightarrow P_1 * P_2$ when P_1 and P_2 pure

$(P_1 \wedge P_2) * P_3 \Leftrightarrow P_1 \wedge (P_2 * P_3)$ when P_1 pure

There is also an inference rule dealing with implication:

$$\frac{P_1 \Rightarrow P_2 \quad P_3 \Rightarrow P_4}{P_1 * P_3 \Rightarrow P_2 * P_4}$$

Another very useful class of propositions are the *domain exact* propositions. A proposition P is called domain exact when for σ and for all f holds that $\sigma \models P$ and $(\sigma \mid f_\sigma \rightarrow f) \models P$ implies that $dom(f_\sigma) = dom(f)$. For domain exact propositions the semidistributive axioms given earlier become fully distributive:

$(P_1 * P_3) \wedge (P_2 * P_3) \Leftrightarrow (P_1 \wedge P_2) * P_3$ when P_3 is domain exact
$(\forall d.\ P_1) * P_2 \Rightarrow \forall d.\ (P_1 * P_2)$ when d not free in P_2 and P_2 is domain exact

Note that all propositions constructed from $\mapsto$, $*$ and quantifiers are domain exact.

Notably absent from this list are two very basic axioms that can often be taken for granted. These are weakening, $P_1 * P_2 \Rightarrow P_1$, and contraction $P \Leftrightarrow P * P$. As a direct result of the exactness inherent in the logic, neither of these axioms hold.

8.2 $\mapsto$ Axioms

Unfortunately, a complete axiom set for $\mapsto$ has not been found. As the focus of this thesis is on the creation of inference rules for program annotation, improving upon this set has not really been looked at.

The first axiom presented in this section is used to 'hide' some information known to be true. An important part of a practically useful logic is a concise notation. In this case, we do not want to have to describe all of an object's fields if we do not know their value. Note that the current syntax and semantics of object notation do not require those fields to be written down, but allow as a smallest possible object notation $O \mapsto []^c$. From an operational point of view it is clear that when an object exists, all fields specified in its class definition will exist as well. This is translated to the following axiom:

$$O \mapsto []^c \Leftrightarrow \exists d.\ O \mapsto [a : d]^c \quad \text{iff} \quad a \in \mathit{ClassFields}(c) \text{ and } O \neq d$$

Remember that *ClassFields* is defined in chapter 4 as the function that gives the set of instance variables of a class c.

The $\Leftarrow$ part of this axiom is almost trivially sound looking at the semantics of $\mapsto$ and $\exists$. Giving a formal proof of the soundness of $\Rightarrow$ however is difficult, but the following arguments should suffice:

- It is clear that to get to a state in which O refers to an object $[]^c$, that object must first have been created;
- There is only one way to create an object, and that is by using a statement $x^{Obj} := \textbf{new } c$;
- The operational semantics of this statement show that when an object is created, all its instance variables, as defined by $\mathit{ClassFields}(c)$, are mapped to a value;
- That is enough to satisfy the semantics of $\exists$, and should be enough to prove the soundness of this axiom.

Note that these arguments are still valid when the left-hand side of the bi-implication would mention some of the instance variables of c, making the axiom above unnecessarily restrictive. A more general version, that is referred to as the *Hiding Axiom*, is:

$O \mapsto [a_1 : E_1^t, \ldots, a_n : E_n^t]^c \Leftrightarrow \exists d.\ O \mapsto [a : d, a_1 : E_1^t, \ldots, a_n : E_n^t]^c$
iff $a \in \mathit{ClassFields}(c)$ and $(\forall i : 1 \leq i \leq n : a \neq a_i)$ and d not in $O \mapsto [a_1 : E_1^t, \ldots, a_n : E_n^t]^c$

Here, $(\forall i : 1 \leq i \leq n : a \neq a_i)$ is needed to satisfy the syntactic obligation not to mention the same instance variable twice.

To improve readability, fields_n is used as a shorthand for $a_1 : E_1^t, \ldots, a_n : E_n^t$ wherever possible.

Now a crucial difference between $*$ and $\wedge$ can be described by the following two axioms:

$O_1 \mapsto [\mathit{fields}_n]^c \wedge O_2 \mapsto [\mathit{fields}_m]^c \Rightarrow O_1 = O_2$
$O_1 \mapsto [\mathit{fields}_n]^c * O_2 \mapsto [\mathit{fields}_m]^c \Rightarrow O_1 \neq O_2$

Note that the first axiom relies on the exactness of the semantics of $\mapsto$, which demands there is only a single location in the domain of the *FieldHeap* function.

One more $\mapsto$ axiom will be given. It expresses in the most general terms that any given instance variable can have only a single value:

$(O_1 \mapsto [a : E_1^t, fields_n]^c * P_1) \wedge (O_1 \mapsto [a : E_2^t, fields_m]^c * P_2) \Leftrightarrow$
$(O_1 \mapsto [a : E_1^t, fields_n]^c * E_1^t \doteq E_2^t * P_1) \wedge (O_1 \mapsto [a : E_2^t, fields_m]^c * P_2)$

8.3 Other Axioms

With $P[E_2^t/E_1^t]$ representing the capture avoiding substitution of E_1^t by E_2^t, an axiom for substitution can be defined:

$$P \wedge E_1^t = E_2^t \Leftrightarrow P[E_2^t/E_1^t] \wedge E_1^t = E_2^t$$

By using structural induction on P, this axiom and the axioms and inference rule from 8.1, a variant can be created where any number of occurrences of E_1^t in a proposition P can be replaced.

Furthermore, two axioms for $\exists$ are introduced:

$P \Leftrightarrow \exists d.\ P$ when d not free in P
$P \Rightarrow \exists d.\ P[d/E^t]$

9 Specifications

This chapter introduces *specifications*, which provide the link between a program and its proof. Proofs are given in the well-known Hoare triple style [3]. A specification is of the shape $\{P_1\}\ S\ \{P_2\}$. Of special interest is what is called 'a form of tight specification' in [9]. This tight specification makes sure that 'well specified programs don't go wrong' [4]. This is accomplished by interpreting a specification in such a way that all resources necessary to safely execute S must be mentioned in P_1 or allocated in S for the specification to hold. The operational semantics allow us to capture this notion of safety as follows:

" $< S, (s, f) >$ is safe" iff $\neg(< S, (s, f) > \rightsquigarrow$ **abort**)

The semantics of specifications are complicated by the use of universally quantified variables. Therefore, a simpler semantics is given first. This semantics treats universally quantified variables as constants to show the essence of the definition:

$\{P_1\}\ S\ \{P_2\}$ holds iff $\forall\sigma : (\sigma \models P_1$ implies
$< S, (s_\sigma, f_\sigma) >$ is safe and
$\forall(s, f) \in OpState : (< S, (s_\sigma, f_\sigma) > \rightsquigarrow\ (s, f)$ implies $(\sigma \mid s_\sigma \rightarrow s, f_\sigma \rightarrow f) \models P_2))$

As can be seen, specifications are quantified implicitly over all states σ, as well as over all possible execution traces (this because of the non-determinism in the operational semantics of the **new** statement). These semantics differ from that of Reynolds and O'Hearn in not needing the notion of a sequence of configuration transitions because of the use of natural semantics in chapter 4 instead of a SOS semantics. Furthermore, the restriction of only dealing with initialized variables was placed on a syntactic level (see chapter 2) instead of on this semantic level. The main reason for this difference is the operational rule for Method Call, where it is convenient to state the *Store* function is empty except for the variable **this**.

The semantics above must be extended to properly incorporate universally quantified variables. The function $FV_D(P)$ is used to give the set of such variables in proposition P.

Then, the semantics of assertions are:

with $FV_D(P_1) = \{D_1, \ldots, D_i\}$:

$\{P_1\}\ S\ \{P_2\}$ holds iff $\forall\sigma : (\forall\delta_1, \ldots, \delta_i : (\sigma' \models P_1$ implies
$< S, (s_\sigma, f_\sigma) >$ is safe and
$\forall(s, f) \in OpState : (< S, (s_\sigma, f_\sigma) > \rightsquigarrow\ (s, f)$ implies $(\sigma' \mid s_{\sigma'} \rightarrow s, f_{\sigma'} \rightarrow f) \models P_2)))$

where $\sigma' = (\sigma \mid u_\sigma(\forall j.\ 1 \leq j \leq i.\ D_j \rightarrow \delta_j))$

Note that s_σ and $s_{\sigma'}$ are equivalent functions (as σ and σ' differ only on the $UniVar$ function).

10 Inference Rules for Program Annotation

This chapter forms the hart of the thesis, satisfying the goal of creating inference rules for specifications. It descibes the inference rules that can be used for program annotation and verification. First, some structural rules are introduced. Then command-specific inference rules are given for all statements in the language. These rules are proven sound in appendix B.

For the non-standard command-specific rules a local version is given first. These almost trivially sound local versions can then be extended to global versions using the frame rule given in the next section. Unlike the rules given by Reynolds and O'Hearn, these global versions are not translated to rules suitable to backwards reasoning. The main reason not to do so is that for backwards reasoning another new operator is required. This operator, called *separating implication*, has the following semantics:

$$\sigma \models P_1 \mathrel{-\!\!*} P_2 \quad \text{iff } \forall f : \text{if } f \# f_\sigma \text{ and } (\sigma \mid f_\sigma \rightharpoonup f) \models P_1 \text{ then } (\sigma \mid f_\sigma \rightharpoonup f_\sigma \cdot f) \models P_2$$

So, for the separating implication $P_1 * P_2$ to hold, a new set of objects that is completly disjoint from the set of objects defined by f_σ in needed. $\sigma \models P_1 * P_2$ represents that, when f defines such a set, and σ, but with the objects as defined by f instead of those defined by f_σ, satisfies P_1, then σ, but with the union of these two sets f and f_σ, satisfies $P_1 * P_2$.

However, as the new operator $*$ is already fairly hard to grasp, we do not want to complicate matters further with an even more difficult to intuitively understand operator. So, although the usefulness of the separating implication is undisputed, we will first try to get as far as possible without it. Therefore, instead of using backwards reasoning, the rules will accommodate a style of forward reasoning, as demonstrated in appendix A.

The inference rules use $[E_2/E_1]$ to represent the capture avoiding substitution of E_1 by E_2. It is used for substitution in both propositions and specifications.

10.1 Frame Rule

When dealing with shared resources, sound local inference rules describing the effect of a statement are often not to hard to provide. However, expanding these rules to global versions is very problematic, as syntactically unrelated expressions can be changed by an action on a shared resource as a result of aliasing. For instance, when x_1 and x_2 refer to the same object, an update of $x_1.a$ will change the value of the syntactically unrelated $x_2.a$ as well.

This logic's solution is to assure a locality for actions involving shared resources by ruling out the possibility that syntactically unrelated statements change. This section presents the rule that takes advantage of this solution, called the frame rule [4]:

$$\frac{\{P_1\}\ S\ \{P_2\}}{\{P_1 * P_3\}\ S\ \{P_2 * P_3\}}$$

Where no temporary variable occurring free in P_3 is modified by S

Here, S is said to modify a variable x^t only when x^t occurs on the left-hand side of an assignment $x^t := E^t$, $x^t := o.a^t$ or $x^t :=$ **new** c in S.

Soundness of this rule is dependent on the semantics of $*$ and the safety demand of specifications. This requires that a valid specification mentions all objects needed for safe execution of its statement in its precondition. A proof of the soundness of this rule can be found in [9].

This rule creates the possibility of making global rules out of local ones. Here, a rule is considered local when it only involves variables and objects that are actually used by the statement S. We can be guided by the operational semantics to see which variables and objects this are.

10.2 Structural Rules

Several other well-known structural rules hold in this logic:

Rule of Consequence:

$$\frac{P_1' \Rightarrow P_1 \quad \{P_1\}\ S\ \{P_2\} \quad P_2 \Rightarrow P_2'}{\{P_1'\}\ S\ \{P_2'\}}$$

Auxiliary Variable Introduction:

$$\frac{\{P_1\}\ S\ \{P_2\}}{\{\exists d.\ P_1\}\ S\ \{\exists d. P_2\}}$$

Conjunction of Assertions:

$$\frac{\{P_1\}\, S\, \{P_2\} \quad \{P_3\}\ S\ \{P_4\}}{\{P_1 \wedge P_3\}\, S\, \{P_2 \wedge P_4\}}$$

Disjunction of Assertions:

$$\frac{\{P_1\}\, S\, \{P_2\} \quad \{P_3\}\ S\ \{P_4\}}{\{P_1 \vee P_3\}\, S\, \{P_2 \vee P_4\}}$$

Variable Substitution:

$$\frac{\{P_1\}\ S\ \{P_2\}}{(\{P_1\}\ S\ \{P_2\})[x_1^t/x_2^t]}$$

where x_1^t does not occur in $\{P_1\}\ S\ \{P_2\}$

10.3 Non-OO rules

This section presents the inference rules for the statements that do not directly use objects. The first of these, the rule for Simple Assignment, is given in a strongest postcondition shape. Therefore, it is slightly more complex than Hoare's standard Assignment Rule [3].

Simple Assignment:

$$\frac{\{P\}}{x^t := e^t}$$
$$\{\exists d.\ x^t = e^t[d/x^t] \wedge P[d/x^t]\}$$

Because of the use of $\wedge$ this rule does not endanger the possible $FieldHeap$ domain exactness, despite the fact that a pure proposition is added.
Capture avoiding substitution is necessary as the value of x^t changes. This might invalidate the relations for x^t as given by P. When x^t is redefined in terms of itself, this rule will reflect the change correctly as can be seen in the example $\{x = 4 \wedge emp\}\ x := x + 1\{\exists d.\ x = d + 1 \wedge (d = 4 \wedge emp)\}$. As $(\exists d.\ x = d + 1 \wedge (d = 4 \wedge emp)) \Rightarrow (x = 5 \wedge emp)$ the Rule of Consequence given above shows the validity of $\{x = 4 \wedge emp\}\ x := x + 1\ \{x = 5 \wedge emp\}$

The other non-OO rules are straightforward:

Sequential Composition:

$$\frac{\{P_1\}\ S_1\ \{P_3\} \quad \{P_3\}\ S_2\ \{P_2\}}{\{P_1\}\ S_1; S_2\ \{P_2\}}$$

Skip:

$$\frac{}{\{P\}\ \mathbf{skip}\ \{P\}}$$

Conditional:

$$\frac{\{P_1 \wedge e^{Bool}\}\, S_1\, \{P_2\} \quad \{P_1 \wedge \neg e^{Bool}\}\, S_2\, \{P_3\}}{\{P_1\}\, \mathbf{if}\ e^{Bool}\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}\, \{P_2 \vee P_3\}}$$

10.4 Object component rules

A very simple local rule can be given for object component assignment $o.a^t := e^t$. Safe execution of the statement is dependent on the existence of the object referred to by o. This object is furthermore required to have a field called a. As a specification is only valid when the execution of the statement is safe, neither emp nor **true** is strong enough as a precondition:

Object Component Assignment, local version:

$$\{o \mapsto [a^t : E^t]^c\}$$
$$o.a^t := e^t$$
$$\{o \mapsto [a^t : e^t]^c\}$$

The global version of this rule, that is a source of problems in a logic without the separating conjunction, is now an almost direct application of the frame rule. This because the effect of this operation is localized by the use of $*$, and because no temporary variables are modified by this statement. However, the separating conjuction localizes the effect of the statement to the object, not to the field. Additional fields of the object remain unchanged, and the rule needs to mention this explicitly. More on the need for this can be found in chapter 11:

Object Component Assignment:

$$\{o \mapsto [a^t : E^t, fields_n]^c * P\}$$
$$o.a^t := e^t$$
$$\{o \mapsto [a^t : e^t, fields_n]^c * P\}$$

Object component lookup, $x^t := o.a^t$, is a combination of the rule for Object Component Assignment and the rule for Simple Assignment. As it modifies x^t, it is necessary to use capture avoiding substitution to replace x^t. A local version is:

Object Component Lookup, local version:

$$\{o \mapsto [a^t : E^t]^c\}$$
$$x^t := o.a^t$$
$$\{\exists d.\ x^t = E^t[d/x^t] \wedge (o \mapsto [a^t : E^t]^c)[d/x^t]\}$$

The global version of this rules again needs to specify the other fields of the object, and account for the fact that x^t might occur in P:

Object Component Lookup:

$$\{o \mapsto [a^t : E^t, fields_n]^c * P\}$$
$$x^t := o.a^t$$
$$\{\exists d.\ x^t = E^t[d/x^t] \wedge (o \mapsto [a^t : E^t, fields_n]^c * P)[d/x^t]\}$$

10.5 Object creation and disposal

The statement $x^{Obj} :=$ **new** c creates a new object of class c and makes x^{Obj} reference that object. Nothing specific is known of the values of the object's instance variables. We use the hiding axiom from 8.2 to simplify the inference rule:

Object Creation, local version:

$$\{emp\}$$
$$x^{Obj} := \textbf{new } c$$
$$\{x^{Obj} \mapsto []^c\}$$

Again, extending to a global version for this rule is simple thanks to the localization provided by the separating conjunction as the object created here is new. The only complication is the change in the value of x^{Obj}, for which substitution with an existentially quantified variable is used again:

Object Creation:

$$\{P\}$$
$$x^{Obj} := \textbf{new } c$$
$$\{\exists d.\ x^{Obj} \mapsto []^c * P[d/x^{Obj}]\}$$

The statement **dispose**(o) is the inverse of the previous statement. A local version of this rule requires o to indeed refer to an object, and the effect of the statement is the removal of that object:

Object Disposal, local version:

$$\{o \mapsto []^c\}$$
$$\textbf{dispose}(o)$$
$$\{emp\}$$

Note that as the entire object is disposed, the values of its instance variables are irrelevant and can be hidden. The frame rule can be applied directly here to create the global version of this rule:

Object Disposal:
$\{o \mapsto []^c * P\}$
dispose(o)
$\{P\}$

10.6 Method Call

Method call is by far the most complex and most ambitious inference rule in this thesis. The only other inference rule that was found for the annotation of a method call that can deal with recursion and call-back is that in [6]. This rule however does not allow for the localization that can be achieved with separation logic.

As a basis for dealing with method call we use the idea that it should not be necessary to prove the body of the method correct every time the method is called. Instead we describe the behavior of a method with two special assertions. The special assertions take the role of pre- and post-condition, and can be referred to as $Pre_{c.m}$ and $Post_{c.m}$ respectively for a method m in class c. The statement that is the body of a method m in a class c can be referred to as $Body_{c.m}$ (which is just a different notation for the function $Body(c, m)$ that was defined in chapter 4). This means that every body of a method has a specification of the form $\{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\}$. Specifications of this form are required to be unprovable when there is no method m in class c, implicitly adding $m \in Methods(c)$ as a requirement to the semantics for such a specification.

Some restrictions are placed on the shape of both $Pre_{c.m}$ and $Post_{c.m}$ due to the operational understanding of the method call as described by the operational semantics (see chapter 4). There it can be seen that execution of a method body always starts in a configuration with a *Store*-function in which only the variable **this** is defined, or in other words, where no temporary variable can have a value. Also, the *Store* function in the terminal state for the execution of the body is not used in the transition for the method call. Therefore it is required that $Pre_{c.m}$ and $Post_{c.m}$ do not contain any temporary variables.

Universally quantified variables were introduced specifically with method call in mind. They provide the means to make the specification of the body globally applicable, and give a way to relate the post-condition to the pre-condition. As an example, the specification
$\{\mathbf{this} \mapsto [a^{Int} : D]^c\}\ Body_{c.m}\ \{\mathbf{this} \mapsto [a^{Int} : (D + 4)\}$ states that the body of method m of class c adds 4 to the value initially in the field a^{Int} of the object referred to by **this**, for any initial value. It is assured by construction that the universally quantified variables in $Post_{c.m}$ are a subset of those in $Pre_{c.m}$ for any valid specification $\{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\}$.

The existence of the Frame Rule (see 10.1) makes it unnecessary for a method's specification to talk about objects not needed for its safe execution. This gives a locality to specifications that is very welcome.

We will not go into further detail on the construction of pre- and postconditions. It is important to note however that one might not be able to prove programs that are operationally correct when the specification of a method the program calls is not strong enough. This means it is extremely important that properties of programs can be expressed in a clear and simple manner.

To get to the final version of our inference rule for method call, a series of sound, and increasingly powerful inference rules is presented below.

Given a valid specification for a method body $\{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\}$ we would like to conclude the validity of the method call triple $\{Pre_{c.m}\}\ o.m\ \{Post_{c.m}\}$ when this call $o.m$ would mean nothing

else than the execution of $Body_{c.m}$. Recall that, once it is proved that this local specification holds, it can be extended by applying the frame rule, creating the global $\{Pre_{c.m} * P\}\ o.m\ \{Post_{c.m} * P\}$. First we need to make sure however that $o.m$ indeed executes method m of class c. We know this to be true when o refers to an object of class c. When this property holds for a proposition P we know that the following implication holds: $P \Rightarrow (o \mapsto []^c * \mathbf{true})$. That means our first attempt at an inference rule looks like:

$$\frac{\{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\} \quad Pre_{c.m} \Rightarrow (o \mapsto []^c * \mathbf{true})}{\{Pre_{c.m}\}\ o.m\ \{Post_{c.m}\}}$$

Method Call, first try:

Unfortunately, $Pre_{c.m} \Rightarrow (o \mapsto []^c * \mathbf{true})$ can only possibly hold in the special case where o is **this** as $Pre_{c.m}$ is the local precondition for the body and may not contain any temporary variables, including o. We are saved by the special variable **this**, which is used as the connection between the body of the method and the object calling the method. More formally, **this** functions as a parameter for the method, giving the body access to the object whose method is called. As can be seen in the operational semantics, variables **this** and o refer to the same object and can be substituted for one another (a process sometimes referred to as context switching). We improve the rule in the following way:

$$\frac{\{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\} \quad Pre_{c.m}[o/\mathbf{this}] \Rightarrow (o \mapsto []^c * \mathbf{true})}{\{Pre_{c.m}[o/\mathbf{this}]\}\ o.m\ \{Post_{c.m}[o/\mathbf{this}]\}}$$

Method Call, second try:

Note that when o refers to an object of class c in $Pre_{c.m}[o/\mathbf{this}]$, **this** refers to an object of class c in $Pre_{c.m}$. As an example of what this rule can prove, assume a method m in a class c for which $Pre_{c.m} = this \mapsto []^c$, $Body_{c.m} = \mathbf{dispose(this)}$ and $Post_{c.m} = emp$.
As $\{this \mapsto []^c\}$ **dispose(this)** $\{emp\}$ holds, $\{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\}$ is a valid specification for the method. We know that $\mathbf{this} \mapsto []^c \Rightarrow \mathbf{this} \mapsto []^c * emp$ and that $emp \Rightarrow \mathbf{true}$. We can now use the frame rule, the hiding axiom for $\mapsto$ and the rule above to see that the specification $\{x_1 \mapsto [a : x_2]^c * x_2 \mapsto [a : nil]^c\}\ x_1.m\ \{x_2 \mapsto [a : nil]^c\}$ holds as well.

The rule has improved, yet several problems remain. One of these problems can be shown by looking at a method for which $Body_{c.m} = \mathbf{skip}$ and $Pre_{c.m} = Post_{c.m} = emp$. Because $Pre_{c.m}$ does not mention **this**, $Pre_{c.m}[o/\mathbf{this}] \Rightarrow (o \mapsto []^c * \mathbf{true})$ will never hold. Despite that, we do expect the specification $\{x \mapsto []^c\}\ x.m\ \{x \mapsto []^c\}$ to hold. We solve this problem by putting the frame rule to good use. The essential point here is that it is not a requirement for the body of $o.m$ that o refers to an object of class c, but a requirement for the precondition of the call. We update the rule to reflect this:

$$\frac{\{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\} \quad (Pre_{c.m}[o/\mathbf{this}] * P) \Rightarrow (o \mapsto []^c * \mathbf{true})}{\{Pre_{c.m}[o/\mathbf{this}] * P\}\ o.m\ \{Post_{c.m}[o/\mathbf{this}] * P\}}$$

Method Call, third try:

The next step deals with the universally quantified variables in a specification of a body. These variables can be seen as formal parameters of the specification and as such can be replaced by actual parameters. The problem of the current rule can be illustrated by looking at a class c with two methods m_1 and m_2. m_1 is specified by (with a and x of type Int):
$\{\mathbf{this} \mapsto [a : D]^c\}\ x := \mathbf{this}.a; \mathbf{this}.a := x + 1\ \{\mathbf{this} \mapsto [a : (D + 1)]^c\}$
This specification can be proven to hold by the rules given so far. Method m_2 is specified by:
$\{\mathbf{this} \mapsto [a : 4]^c\}\ \mathbf{this}.m_1\{\mathbf{this} \mapsto [a : 5]^c\}$
As we have no axiom for introducing universally quantified variables (and do not intent to create one), the implication $\mathbf{this} \mapsto [a : 4]^c \Rightarrow \mathbf{this} \mapsto [a : D]^c$ does not hold, which means the rule of consequence cannot be applied, making it impossible to prove this specification which we know to

be operationally correct.
The solution lies in the universal nature of the variable D. This means we can substitute any other expression for D in a specification and it still holds as long as the value of the expression is not changed by the specification's statement (as the value of D is not changed by the specification either). The most general rule we can formulate to express this is:

D-Elimination rule, general version:

$$\frac{\{P_1\}\ S\ \{P_2\}}{\{P_1[E^t/D]\}\ S\ \{P_2[E^t/D]\}}$$

Where S does not modify the value of E^t.

Note that allowing expressions of a form other than E^t would be useless as substituting them would certainly lead to a syntactically incorrect proposition. Instead of trying to specify when a statement modifies an expression's value in general, we use a weaker rule that is more directly related to the problem at hand:

D-Elimination rule for method call:

$$\frac{\{P_1\}\ o.m\ \{P_2\}}{\{P_1[E^t/D]\}\ o.m\ \{P_2[E^t/D]\}}$$

As is shown in the soundness proof for this rule (see B.5), an expression E^t is guaranteed not to be changed by a method call $o.m$. Informally, this is true because $o.m$ only changes the $FieldHeap$ function, while E^t is independent of that function.

Given the valid specification for method m_1 we can now prove the specification given for method m_2, as we can first prove $\{\mathbf{this} \mapsto [a : D]^c\}\ \mathbf{this}.m_1\{\mathbf{this} \mapsto [a : (D+1)]^c\}$ and then use the D-elimination rule with 4 as E^t. Some other examples of specifications now provable are (with $y \in TempVar^{Int}$):
$\{\mathbf{this} \mapsto [a : (x+y)]^c\}\ \mathbf{this}.m_1\{\mathbf{this} \mapsto [a : (x+y+1)]^c\}$
and
$\{\exists d.\ \mathbf{this} \mapsto [a : d]^c\}\ \mathbf{this}.m_1\{\exists d.\ \mathbf{this} \mapsto [a : (d+1)]^c\}$

To reach the final version of our inference rule for method call we deal with two more important issues. As can be seen in the previous versions of the rule, we can only prove that a specification for a method call holds if we can prove that the specification for that method's body holds. But if the proof of the body's specification is in turn dependent on the proof of the method call, as is the case when recursion or call-back enter the picture, this proof can never be constructed as the proof of the body would be dependent on itself.
We deal with this problem in an induction-like manner, breaking the mutual dependency by allowing that the proof of the body's specification may assume the correctness of the method call's specification. The symbol $\vdash$ is introduced for such assumptions, where, with A a set of inference rules, $A \vdash \{P_1\}\ S\ \{P_2\}$ holds if we can prove the specification $\{P_1\}\ S\ \{P_2\}$, using the inference rules specified in A if necessary. The assumption needed for recursion and call-back is:

Assumption for recursion and call-back:

$$\frac{(Pre_{c.m}[o'/\mathbf{this}] * P') \Rightarrow (o' \mapsto []^c * \mathbf{true})}{\{Pre_{c.m}[o'/\mathbf{this}] * P'\}\ o'.m\ \{Post_{c.m}[o'/\mathbf{this}] * P'\}}$$

Here, P' and o' are used to distinguish from the P and o in the next rule because, with A the singleton set of this inference rule, we can update the inference rule for method call once again:

Method Call with Recursion:

$$\frac{\begin{array}{c}A \vdash \{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\} \\ (Pre_{c.m}[o/\mathbf{this}] * P) \Rightarrow (o \mapsto []^c * \mathbf{true})\end{array}}{\{Pre_{c.m}[o/\mathbf{this}] * P\}\ o.m\ \{Post_{c.m}[o/\mathbf{this}] * P\}}$$

11 Design Decisions

The previous chapter completes the formal description of the logic, ending with and building up to the inference rules. In this chapter, some of the design decisions made are further explained.

The focus is on the decision not to allow a proposition to talk exclusively about the smallest shared resource in an OO-language, a field, or instance variable, of an object. Instead, an entire object must be mentioned at once. This might seem strange as sound local rules seem to indicate that this is a possibility:

$$\begin{array}{c} \{o.a^t = E^t\} \\ o.a^t := e^t \\ \{o.a^t = e^t\} \end{array} \qquad \begin{array}{c} \{o.a^t = E^t\} \\ x^t := o.a^t \\ \{\exists d.\ x^t = E^t[d/x^t] \wedge (o.a^t = E^t)[d/x^t]\} \end{array}$$

There are some notational problems with the rules above. Central to the logic is that a shared resource is not allowed to appear in a proposition more than once. It is this feature that allows global rules for statements that modify such resources in the first place. When a dereferencing operator were allowed in an equality, substitution would create unwanted situations. As an example, the implication $(x_1.a = 4 * x_2.a = 4) \Rightarrow (x_1.a = x_2.a * x_2.a = 4)$ is very much unwanted (as an update of $x_2.a$ would now incorrectly imply a change in the value of $x_1.a$). However, these problems are easily solved by the introduction of the $\mapsto$ operator and are not the reason for disallowing the field as the smallest shared resource.

In fact, one could get quite far with a model that is a fairly straightforward extension of the models presented in [8] and [9]. In such a model the *FieldHeap* has the following definition:

$$FieldHeap \ : \ \mathcal{A} \times InstVar \rightarrow_{par} \mathcal{D}$$

There are several advantages to this model. First of all, simpler inference rules are possible for the Object Component rules:

$$\begin{array}{c} \{o.a^t \mapsto E^t * P\} \\ o.a^t := e^t \\ \{o.a^t \mapsto e^t * P\} \end{array} \qquad \begin{array}{c} \{o.a^t \mapsto E^t * P\} \\ x^t := o.a^t \\ \{\exists d.\ x^t \doteq E^t[d/x^t] * (o.a^t \mapsto E^t * P)[d/x^t]\} \end{array}$$

Secondly, pre- and post-conditions for methods can be even further localized. As an example, consider a class c with instance variables a and b. Now consider a method m of c that sets a to 4. A very logical specification for such a method in the logic presented in this thesis would seem to be:
$\{Pre_{c.m} : \mathbf{this} \mapsto []^c\}\ \mathbf{this}.a := 4\{Post_{c.m} : \mathbf{this} \mapsto [a : 4]^c\}$
However, while this specification is valid, it is not as strong as should be. This can be seen by looking at:
$\{x \mapsto [a : 3, b : 2]^c\}\ x.m\ \{\ x \mapsto [a : 4, b : 2]\}$
This specification cannot be proven valid using the rules given so far. The problem is that method m does not guarantee that the value of b is unchanged. To correct this, the specification of the method must be changed to:
$\{Pre_{c.m} : \mathbf{this} \mapsto [b : D]^c\}\ \mathbf{this}.a := 4\{Post_{c.m} : \mathbf{this} \mapsto [a : 4, b : D]^c\}$.

With a *FieldHeap* as defined above, method m could have a specification along the lines of
$\{Pre_{c.m} : \mathbf{this}.a \mapsto D\}\ \mathbf{this}.a := 4\ \{\mathbf{this}.a \mapsto 4\}$
Now a specification:
$\{x.a \mapsto 3 * x.b \mapsto 2\}\ x.m\ \{x.a \mapsto 4 * x.b \mapsto 2\}$
should be proveable when x refers to an object c as there is a separating conjunction between the precondition and $x.b \mapsto 2$.

The paragraph above hints at one of the advantages of the current logic: it provides a very nice

way to present not only the static type of a variable, but the dynamic type, the type of the object it refers to, as well. Another advantage is that the data hiding axiom as defined in 8.2 does not have a anolog in the logic with the *FieldHeap* as defined in this chapter. For example, while $o \mapsto [a : E_1, b : E_2]^c \Rightarrow o \mapsto [a : E_1]^c$ holds, $(o.a \mapsto E_1 * o.b \mapsto E_2) \Rightarrow o.a \mapsto E_1$ does not.

The main reason to use the current version of the *FieldHeap*, wich is a fairly straightforward extension of the *Heap* function as defined in [4], is that it allows one to be exact in the number of *objects* that are defined, rather than the number of *fields*. This difference is of special importance when there are statements that act on an entire object at once. As it happens, one such statement is defined. This statement is **dispose**. Note that while [8] and [9] both allow the creation of *cons* cells of varying length, it is not possible to dispose of such a construction with a single statement. The analog to the situation here is that instead of a statement **dispose**(o) that disposes of the object referred to by o there would be a statement **dispose**($o.a$) that disposes of a single field. However, such an operation changes the structure of an object in such a way that it no longer meets its class specification.

As to how far the two versions can be made equivalent, for instance by introducing smart shorthand notations, was not looked into. However, the current version at the very least feels more natural by giving a central role to objects in an object-oriented language.

12 Extensions

This chapter extends both the programming language and the assertion language, allowing for more advanced programs to be constructed and proved.

12.1 Input Parameters

One very useful extension of the programming language is the extension of method calls with parameters. While in the current logic instance variables can be used to mimic parameters, this technique is quite cumbersome. Luckily, parameters can be incorporated in a fairly straightforward manner. This is mainly due to the fact that the current language already does have one parameter of a method call, this being the special variable **this**. Therefore, extending with other pass-by-value parameters can be done in a very similar fashion.

First, the set of formal parameters $FPar$, typical element p, is introduced. The set consists of the union of the two sets $FPar^{Int}$ and $FPar^{Obj}$, typical elements p^{Int} and p^{Obj} respectively.

The set $ObjectRef$ is extended with $FPar^{Obj}$ and the set Exp^{Int} with $FPar^{Int}$.

Then, the language grammar of chapter 2 is changed. Instead of the statement $o.m$ the statement $o.m(E_1^t, \ldots, E_n^t)$ is introduced, where $E_1^t, \ldots, E_n^t$ are the *actual parameters* of the method call. Furthermore, the definition of $MethodDeclaration$ is changed to:

$$\mu \quad ::= \textbf{method}\ m\ (p_1, \ldots, p_n)\ \{S\}$$

Here, $p_1, \ldots, p_n$ are the *formal parameters* of the method and the statement S.

Now, the notion of a statically correct program declaration (see chapter 2) is extended by changing the last requirement from:

- Every temporary variable occurring on the right-hand side of an assignment in a statement S must occur earlier on the left-hand side of an assignment in that same statement S

to:

- Every temporary variable that occurs on the right-hand side of an assignment *or is used as an actual parameter* in a statement S must occur earlier on the left-hand side of an assignment in that same statement S.

and adding:

- Every actual parameter corresponds to a formal parameter of the same type

The definition of $Store$ (see chapter 3) is extended so it maps elements of $FPar$ to $\mathcal{D}$ as well, and the valuation $|[e^t]|_s$ is similarly updated.

The operational rule for method call (rule number 5 in chapter 4) now needs access to the names of the formal parameters of a method. To this end, the function $Param$ is introduced, where $Param(c, m, i)$ returns the i-th parameter of method m of class c:

$$Param : ClassName \times MethodName \times IntLit \rightarrow_{par} FPar$$

The non-aborting rule becomes:

$$\frac{\begin{array}{c} o \in dom(s) \quad s(o) \in dom(f) \quad m \in Methods(f(s(o))\rfloor_1) \\ s'' = (initS \mid \textbf{this} \rightarrow s(o), \forall i : 1 \leq i \leq n : Param(f(s(o))\rfloor_1, m, i) \rightarrow E_i^t) \\ < Body(f(s(o))\rfloor_1, m), (s'', f) > \rightsquigarrow\ (s', f') \end{array}}{< o.m(E_1^t, \ldots, E_n^t), (s, f) > \rightsquigarrow\ (s, f')}$$

The aborting rule is changed in the same way.

The valuation of $|[E^t]|_\sigma$ is updated to map a formal parameter to a value using the *Store* function.

This way, parameters behave in the exact same way as **this**. Therefore, the rule for method call can be updated to:

$$\text{Method Call, parameters:}$$
$$\frac{\begin{array}{c}A \vdash \{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\} \\ P_1 \Rightarrow (o \mapsto []^c * \mathbf{true})\end{array}}{\begin{array}{c}A' \vdash \{Pre_{c.m}[o/\mathbf{this}, E_1^t/Param(c,m,1), \ldots, E_n^t/Param(c,m,n)] * P\} \\ o.m(E_1^t, \ldots, E_n^t) \\ \{Post_{c.m}[o/\mathbf{this}, E_1^t/Param(c,m,1), \ldots, E_n^t/Param(c,m,n)] * P\}\end{array}}$$

Where P_1 is defined by $Pre_{c.m}[o/\mathbf{this}, E_1^t/Param(c,m,1), \ldots, E_n^t/Param(c,m,n)] * P$

As an interesting aside, when the suggestions from chapter 10.7 are implemented, parameters can also be seen as variations on universally quantified logical variables. A much simpler rule can be formulated:

$$\text{Call Safe Parameter Elimination:}$$
$$\frac{\{P_1\}\ c.m\ \{P_2\}}{\{P_1[E^t/p^t]\}\ c.m\ \{P_2[E^t/p^t]\}}$$

And important factor that allows for the construction of these rules is that the value of a parameter cannot be changed by the program.

12.2 Method Output

Another nice extension that is easily added is the addition of a return value to methods.

To accomplish this, the set $MethodNames$ is constructed from the two sets $MethodNames^{Int}$ and $MethodNames^{Obj}$.
The statement $o.m$, defined in the language grammar (chapter 2) is replaced with the statement $x^t := o.m^t$.
In every proposition $Post_{c.m}$ the local variable $\mathbf{result}^t$ is required to occur. It must have the same type as method m of class c.
$\mathbf{result}^t$ represents the return value of the method. A slight change in the operational semantics for method call (rule 5 from chapter 4) reflects this:

$$\frac{\begin{array}{c}o \in dom(s) \quad s(o) \in dom(f) \quad m \in Methods(f(s(o))\rfloor_1) \\ < Body(f(s(o))\rfloor_1, m), ((initS \mid \mathbf{this} \rightarrow s(o), f) > \rightsquigarrow\ (s', f') \\ \mathbf{result}^t \in dom(s')\end{array}}{< x^t := o.m^t, (s, f) > \rightsquigarrow\ ((s \mid x^t \rightarrow s'(\mathbf{result}^t), f')}$$

The inference rule for method call can now be changed to:

$$\text{Method Call, return value:}$$
$$\frac{\begin{array}{c}A \vdash \{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\} \\ (Pre_{c.m}[o/\mathbf{this}] * P) \Rightarrow (o \mapsto []^c * \mathbf{true})\end{array}}{A' \vdash \{Pre_{c.m}[o/\mathbf{this}] * P\}\ x^t := o.m^t\ \{\exists d.\ Post_{c.m}[o/\mathbf{this}, x^t/\mathbf{result}] * P[d/x^t]\}}$$

As a final note, as this statement modifies x^t, x^t can't be used in the rule for D-Elimination.

12.3 Abstract Values - Logical Lists

For many interesting functions it is necessary to not only be able to relate the pre- and post-condition in terms of structure, but also to relate the abstract values represented by these structures. An important abstract value is the logical list (also known as sequence) of model values. Let List of $\mathcal{D}$ be the set of such lists.

Quantified versions of this abstract value are represented in the Assertion Language Grammar (see chapter 5) by two additional sets, analog to the sets $ExiVar$ and $Univar$ that represent quantified model values. The first is the set $ExiListVar$, typical element l, and the second $UniListVar$, typical element L. Two functions are defined to map these values to logical lists:

$$\begin{array}{lcl} ExiList & : & ExiListVar \rightarrow_{par} \text{List of } \mathcal{D} \\ UniList & : & UniListVar \rightarrow_{par} \text{List of } \mathcal{D} \end{array}$$

The assertion language grammar is extended with the set of logical lists $LogList$, that has typical element λ and is defined by:

$$\lambda ::= \varepsilon \mid [E^t] \mid l \mid L \mid \lambda_1 +\!\!+ \lambda_2$$

The relation between λ and List of $\mathcal{D}$ can be given by a straightforward valuation when the $AsState$ tuple is extended with functions $ExiList$ and $UniList$. This valuation $|[\lambda]| : AsState \rightarrow_{par} \text{List of } \mathcal{D}$ will have ε represent the empty list, $[E^t]$ represent the singleton list of the model value represented by E^t and $\lambda_1 +\!\!+ \lambda_2$ represent the concatenation of two lists.

The set of propositions $AProp$ defined in chapter 5 is extended with list equality and existential quantification over lists:

$$P ::= \ldots \mid \lambda_1 = \lambda_2 \mid \exists l.\ P$$

The semantics of these new propositions are straightforward, for example:

$$\sigma \models \lambda_1 = \lambda_2 \quad \text{iff} \quad |[\lambda_1]|_\sigma = |[\lambda_2]|_\sigma$$

The simplest program structure to implement a logical list in is the singly linked list. Such a structure can be described in a general way by the predicate:

$$List : LogList \times AExp^{Obj} \times InstVar \times InstVar \times ClassName \rightarrow AProp$$

$List.\lambda.E^{Obj}.a_1.a_2.c$ represents that the logical list λ is implemented in fields a_1 of a list of class c that is linked by fields a_2, and that E^{Obj} is an entry into this list. This predicate can be defined with structural induction on the logical list:

$$\begin{array}{lcl} List.\varepsilon.E^{Obj}.a_1.a_2.c & \stackrel{def}{=} & emp \wedge E^{Obj} = \mathbf{nil} \\ List.([E^t] +\!\!+ \lambda).E^{Obj}.a_1.a_2.c & \stackrel{def}{=} & \exists d.\ E^{Obj} \mapsto [a_1 : E^t, a_2 : d]^c * List.\lambda.d.a_1.a_2.c \end{array}$$

As a variation, the predicate $ListRng$ represent a list segment from E_1^{Obj} to E_2^{Obj} when defined by:

$$\begin{array}{lcl} ListRng.\varepsilon.E_1^{Obj}.E_2^{Obj}.a_1.a_2.c & \stackrel{def}{=} & emp \wedge E_1^{Obj} = E_2^{Obj} \\ ListRng.([E^t] +\!\!+ \lambda).E_1^{Obj}.E_2^{Obj}.a_1.a_2.c & \stackrel{def}{=} & \exists d.\ E_1^{Obj} \mapsto [a_1 : E^t, a_2 : d]^c * \\ & & ListRng.\lambda.d.E_2^{Obj}.a_1.a_2.c \end{array}$$

The use of the $*$ here guarantees that there are no cycles within the list segment.

Useful properties of the $List$ predicate include:

$List.\lambda.E^{Obj}.a_1.a_2.c \Rightarrow (E^{Obj} = \mathbf{nil} \Rightarrow \lambda = \varepsilon)$
$List.\lambda.E^{Obj}.a_1.a_2.c \Rightarrow (E^{Obj} \neq \mathbf{nil} \Rightarrow (\exists d, l_0.\ \lambda = [d] +\!\!+ l_0))$

L can be used in the same way as D, which requires an change in the semantics of assertions that is not given here. What is given is the inference rule:

L–Elimination for method call:

$$\frac{\{P_1\}\ o.m\ \{P_2\}}{\{P_1[\lambda/L]\}\ o.m\ \{P_2[\lambda/L]\}}$$

Appendix A.1 presents an example that uses the $List$ predicate and a method with output as described in the previous section to prove the specification of a program that calculates the sum of the fields of a linked list by means of recursion.

13 Conclusions

Although much much work remains to be done, separation logic most certainly seems a useful tool for reasoning about object oriented programs. The localization provided by $*$ makes it possible to reason effectively about methods in isolation, and then provides the means to prove specifications of calls to these methods from the surrounding environment.

This thesis introduces a simple object oriented programming language, and provides an operational semantics to formally capture how a program in the language is executed. It introduces an assertion language that captures properties of states of the program. This assertion language is based on separation logic, but has the object as its smallest shared resource. Axiomatic reasoning about statements and their effect is done using specifications, that are based on the well-known Hoare triples.

The main goal of of the project is reached by the creation of global inference rules for program annotation. The soundness of these rules is proved. Combining statement-specific rules with structural rules allows for the verification of programs of practical value, as is shown by several examples.

Of special interest is the the construction of an simple yet powerful inference rule for method call annotation that accounts for both recursion and call-back. The explicit capturing of both static and dynamic types of variables plays an implortant role here.

As stated, much remains to be done. It will be interesting to see how the proof system's capabilities are effected by the introduction of a proper subtyping relation which might require the generalization of the dynamic types. Furthermore, this thesis only looked into partial correctness. Note however that the exactness inherent in the logic gives an extra handle for proving termination. Other areas of future research could include further extensions to the abstract values and the implementations of such values that can be reasoned about. Then there is another very essential part of the object orientation methodology which deals with interfacing and data-hiding that is currently not accounted for.

Of great importance is also the goal of automated verification. To this end, clear rules that give a direction to a proof are required. The locality and scalability of the rule for method call might be a step in the right direction. However, no explicit work in this area was done.

References

[1] F.S. de Boer. A wp-calculus for OO. In W. Thomas, editor, *Proceedings of the Second International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '99), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS '99), (Amsterdam, the Netherlands, April 1999)*, volume 1578 of *Lecture Notes in Computer Science*, pages 135-149. Springer-Verlag, 1999.

[2] R.M. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 23-50. Edinburgh University Press, Edinburgh, Scotland, 1972.

[3] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576-580 and 583, October 1969.

[4] S. Ishtiac and P.W. O'Hearn. BI as an assertion language for mutable data structures. In *Conference Record of POPL 2001: the 28th ACM SIGPLAN-SIGACT Symposium on Principals of Programming Languages*, pages 14-26, New York, 2001. ACM.

[5] P.W. O'Hearn, J.C. Reynolds and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourgh, editor, *Computer Science Logic*, Volume 2142 of *Lecture Notes in Computer Science*, pages 1-19, Berlin, 2001. Springer-Verlag.

[6] A. Poetzsch-Heffter and P. Muller. *A programming logic for sequential Java.* In S.D. Swierstra, editor, Programming Languages and Systems, volume 1576 of *Lecture Notes in Computer Science*, pages 162-176. Springer-Verlag, 1999.

[7] J.C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In J. Davies, B. Roscoe and J. Woodcock, editors, *Milennial Perspectives in Computer Science*, pages 303-321, Houndsmill, Hampshire, 2000. Palgrave.

[8] J.C. Reynolds. Separation Logic: A Logic For Shared Mutable Data Structures. In *Third Annual Symposium on Logic in Computer Science*, Copenhagen, Denmark, 2002. IEEE Computer Society.

[9] H. Yang and P.W. O'Hearn. A semantic basis for local reasoning. In M. Nielsen and U. Engberg, editors, *Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 402-416, Berlin, 2002. Springer-Verlag.

A Examples

To prevent confusion, $<$ is used instead of $\{$ in the annotation of the problem specifications. Furthermore, by writing $< L1 : P >$, proposition P is labeled $L1$, and the two will be considered interchangable from that point onwards in a specification or proof. In other words, this labeling adds a property $L1 \Leftrightarrow (P)$ to the proof system that will be referred to as def $L1$.

To focus on the proof of the methods, **main** is omitted from problem specifications.

A.1 LSum

In this example the specification of a method that calculates the sum of all data fields of a linked list is proven. As the method is recursive, the inference rule for method call annotation is put to good use here. This example also demonstrates the extensions introduced in 12.3 and 12.2, by using the *List* predicate in the specification of the method and by having the method return a value.

Problem Specification
Assume that Node $\in ClassName$, val $\in IntVar^{Int}$ and next $\in IntVar^{Obj}$
Two new predicates can now be defined. The first, $NList$, is little more then a shorthand notation for a *List* of class Node with data fields val that are linked by fields next.
The second, *Sum*, gives the sum of a logical list and is defined by using structural induction on that list:

$$NList.\lambda.E^{obj} \quad \stackrel{def}{=} \quad List.\lambda.E^{obj}\text{.val.next.Node}$$

$$Sum.\varepsilon \quad \stackrel{def}{=} \quad 0$$
$$Sum.([E^{Int}] +\!\!+ \lambda) \quad \stackrel{def}{=} \quad E^{Int} + Sum.\lambda$$

Then, with LSum $\in MethodName^{Int}$, x,y $\in TempVar^{Int}$ and z $\in TempVar^{Obj}$, the problem specification is:

```
Class Node { val, next;
  Method LSum {
    < Pre_Node.LSum : this ↦ [val: D0, next: D1]^Node * NList.L.D1 >
    x := this.val
    ;z := this.next
    ;if z = nil then
      result := x
    else
      y := z.LSum
      ;result := x+y
    fi
    < Post_Node.LSum : Pre_Node.LSum ∧ result = Sum.([D0] ++ L) >
  }
}
```

An annotation
Below a fully and correctly annotated version of the method is given. This annotation can be obtained in a very straigtforward way and can be used as a proof of the correctness of the problem specification. Implicitly used are commutativity and associativity of $*$ and $\wedge$, as well as the axiom: $P_1 \wedge (P_2 * P_3) \Leftrightarrow (P_1 \wedge P_2) * P_3$ when P_1 is pure, wich can be found in 8.1. Then the annotation is a direct result of the application of the inference rules for statement annotation from chapter 10, except for two interesting conclusions labeled $B1$ and $B2$. Their correctness is proved seperately below. The annotated version is:

Method LSum {
$< Pre_{Node.LSum} : \textbf{this} \mapsto [\text{val: } D_0, \text{ next} : D_1]^{Node} * NList.L.D_1 >$
x := **this**.val
$< x = D_0 \wedge Pre_{Node.LSum} >$
;z := **this**.next
$< z = D_1 \wedge x = D_0 \wedge Pre_{Node.LSum} >$
;**if** z = **nil then**
$< z = \textbf{nil} \wedge z = D_1 \wedge x = D_0 \wedge Pre_{Node.LSum} >$
result := x
$< L1 : \textbf{result} = x \wedge z = \textbf{nil} \wedge z = D_1 \wedge x = D_0 \wedge Pre_{Node.LSum} >$
else
$< z \neq \textbf{nil} \wedge z = D_1 \wedge x = D_0 \wedge Pre_{Node.LSum} >$
y := z.LSum
$< B1 : \exists d_0, d_1, l_0.\ ((z \mapsto [\text{val: } d_0, \text{ next: } d_1]^{Node} * NList.l_0.d_1) \wedge y = Sum.(d_0 +\!\!+ l_0)) *$
$(\textbf{this} \mapsto [\text{val: } D_0, \text{ next: } D_1]^{Node} \wedge L = ([d_0] +\!\!+ l_0) \ \wedge z = D_1 \wedge x = D_0) >$
;**result** := x+y
$< L2 : \textbf{result} = x + y \wedge B1 >$
fi
$< B2 : Post_{Node.LSum} : Pre_{Node.LSum} \wedge \textbf{result} = Sum.(D_0 +\!\!+ L) >$
}

The Method Call Specification
The most interesting specification in the annotated method above reads:
$\{z \neq \textbf{nil} \wedge z = D_1 \wedge x = D_0 \wedge Pre_{Node.LSum}\}$ y := z.LSum $\{B_1\}$

This specification needs to be proven using the inference rule for method call with return value:

$$\frac{\begin{array}{c}\text{Method Call, return value:} \\ A \vdash \{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\} \\ (Pre_{c.m}[o/\textbf{this}] * P) \Rightarrow (o \mapsto []^c * \textbf{true})\end{array}}{A' \vdash \{Pre_{c.m}[o/\textbf{this}] * P\}\ x^t := o.m^t\ \{\exists d.\ Post_{c.m}[o/\textbf{this}, x/\textbf{result}] * P[d/x^t]\}}$$

Two conclusions can be drawn:
1. The set A' is empty
2. The application of this rule requires the specification for the body of the method to be valid, which already was the proof obligation. However, this may now be done under the assumptions in set A, which contains exactly one assumption that reads:

$$\frac{\begin{array}{c}\text{Assumption for recursion and call-back:} \\ (Pre_{Node.LSum}[o'/\textbf{this}] * P') \Rightarrow (o' \mapsto []^{Node} * \textbf{true})\end{array}}{\{Pre_{Node.LSum}[o'/\textbf{this}] * P'\}\ o'.m\ \{Post_{Node.LSum}[o'/\textbf{this}] * P'\}}$$

Below is a proof tree for the method call's specification, followed by an explanation of its correctness. The proof tree uses the following label:

$P1$: $\textbf{this} \mapsto [\text{val: } D_0, \text{ next: } D_1]^{Node} \wedge L = ([d_0] +\!\!+ l_0) \wedge z = D_1 \wedge x = D_0$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{Pre_{Node.LSum}[z/\mathbf{this}] \;\Rightarrow\; z \mapsto []^{Node} * \mathbf{true}}\,(2)}}{\{Pre_{Node.LSum}[z/\mathbf{this}]\}\; y := z.LSum\; \{Post_{Node.LSum}[z/\mathbf{this}, y/\mathbf{result}]\}}\,A}}{\{z \mapsto [\text{val: } D_0, \text{ next: } D_1]^{Node} * NList.L.D_1\}\; y := z.LSum\; \{(z \mapsto [\text{val: } D_0, \text{ next: } D_1]^{Node} * NList.L.D_1) \wedge y = Sum.(D_0 +\!\!+ L)\}}\,def\ Pre/Post}}{\{z \mapsto [\text{val: } d_0, \text{ next: } d_1]^{Node} * NList.L.d_1\}\; y := z.LSum\; \{(z \mapsto [\text{val: } d_0, \text{ next: } d_1]^{Node} * NList.L.d_1) \wedge y = Sum.(d_0 +\!\!+ L)\}}\,D\text{-elim, 2x}}{\{z \mapsto [\text{val: } d_0, \text{ next: } d_1]^{Node} * NList.l_0.d_1\}\; y := z.LSum\; \{(z \mapsto [\text{val: } d_0, \text{ next: } d_1]^{Node} * NList.l_0.d_1) \wedge y = Sum.(d_0 +\!\!+ l_0)\}}\,L\text{-elim}}{\{(z \mapsto [\text{val: } d_0, \text{ next: } d_1]^{Node} * NList.l_0.d_1) * P1\}\; y := z.LSum\; \{((z \mapsto [\text{val: } d_0, \text{ next: } d_1]^{Node} * NList.l_0.d_1) \wedge y = Sum.(d_0 +\!\!+ l_0)) * P1\}}\,\text{Frame Rule}}{\{\exists d_0, d_1, l_0.\ (z \mapsto [\text{val: } d_0, \text{ next: } d_1]^{Node} * NList.l_0.d_1) * P1\}\; y := z.LSum\; \{\exists d_0, d_1, l_0.\ ((z \mapsto [\text{val: } d_0, \text{ next: } d_1]^{Node} * NList.l_0.d_1) \wedge y = Sum.(d_0 +\!\!+ l_0)) * P1\}}\,\text{AV-intro}}{\{z \neq \mathbf{nil} \wedge z = D_1 \wedge x = D_0 \wedge Pre_{Node.LSum}\}\; y := z.LSum\; \{\exists d_0, d_1, l_0.\ ((z \mapsto [\text{val: } d_0, \text{ next: } d_1]^{Node} * NList.l_0.d_1) \wedge y = Sum.(d_0 +\!\!+ l_0)) * P1\}}\,ROC, (1)$$

Perhaps the most important step in this deduction is the first one. This application of the Rule of Consequence shows the relation between the precondition that was established and the most local correct precondition for a call $z.LSum$. It establishes what will and will not be changed by the call. Crucial is the following property of the *List* predicate as already stated in 12.3:

$$LProp1 \quad : \quad List.\lambda.E^{Obj}.a_1.a_2.c \Rightarrow (E^{Obj} \neq \mathbf{nil} \Rightarrow (\exists d, l_0.\ \lambda = [d] +\!\!+ l_0))$$

This property is referred to as $LProp1$.
The following fairly detailed series of implications shows the Rule of Consequence can indeed be applied in this way:

(1) $z \neq \mathbf{nil} \wedge z = D_1 \wedge x = D_0 \wedge Pre_{Node.LSum}$
$\Leftrightarrow$ {Substitution: $z = D_1$}
$D_1 \neq \mathbf{nil} \wedge z = D_1 \wedge x = D_0 \wedge Pre_{Node.LSum}$
$\Leftrightarrow$ {def $Pre_{Node.LSum}$}
$D_1 \neq \mathbf{nil} \wedge z = D_1 \wedge x = D_0 \wedge (\mathbf{this} \mapsto [\text{val: } D_0, \text{ next: } D_1]^{Node} * NList.L.D_1)$
$\Leftrightarrow$ {$LProp1$ using $NList.L.D_1$ and $D_1 \neq nil$}
$\exists d_0, l_0.\ L = ([d_0] +\!\!+ l_0) \wedge D_1 \neq \mathbf{nil} \wedge z = D_1 \wedge x = D_0 \wedge (\mathbf{this} \mapsto [\text{val: } D_0, \text{ next: } D_1]^{Node} * NList.L.D_1)$
$\Rightarrow$ {$(P \wedge D_1 \neq nil) \Rightarrow P$}
$\exists d_0, l_0.\ L = ([d_0] +\!\!+ l_0) \wedge z = D_1 \wedge x = D_0 \wedge (\mathbf{this} \mapsto [\text{val: } D_0, \text{ next: } D_1]^{Node} * NList.L.D_1)$
$\Leftrightarrow$ {Substitution: $L = ([d_0] +\!\!+ l_0)$; def $NList$}
$\exists d_0, l_0.\ L = ([d_0] +\!\!+ l_0) \wedge z = D_1 \wedge x = D_0 \wedge (\mathbf{this} \mapsto [\text{val: } D_0, \text{ next: } D_1]^{Node} *$
$(\exists d_1.\ D_1 \mapsto [\text{val: } d_0, \text{ next: } d_1]^{Node} * NList.l_0.d_1)$
$\Leftrightarrow$ {Substitution: $z = D_1$}
$\exists d_0, d_1, l_0.\ z \mapsto [\text{val: } d_0, \text{ next: } d_1]^{Node} * NList.l_0.d_1 * (\mathbf{this} \mapsto [\text{val: } D_0, \text{ next: } D_1]^{Node} \wedge L = ([d_0] +\!\!+ l_0) \wedge$
$z = D_1 \wedge x = D_0)$

Then, after an application of the rule for Auxiliary Variable Introduction (see 10.2), the Frame Rule (10.1) can be applied. This is possible because $y := z.LSum$ only modifies y, which does not occur in $P1$.

After introducing L using the rule for $L-$elimination from 12.3 and D_0 and D_1 using $D-$elimination as introduced in 10.6, **this** and **result** are introduced by the definition of pre- and post-condition of the method. Using the assumption in A, we know this is a valid specification when z does indeed refer to an object of class $Node$ in the precondition. That this is the case is shown by:

$$
\begin{array}{ll}
(2) & Pre_{Node.LSum}[z/this] \\
\Leftrightarrow & \{\text{def } Pre_{Node.LSum}[z/this]\} \\
 & z \mapsto [\text{val: } D_0, \text{ next: } D_1]^{Node} * NList.L_0.D_1 \\
\Rightarrow & \{P \Rightarrow \textbf{true}\} \\
 & z \mapsto []^{Node} * \textbf{true}
\end{array}
$$

This concludes the proof of the validity of the specification given in the annotated method.

Establishing the postcondition

The other interesting conclusion in the annotated method is the point labeled $B2$, where the post-condition is established. To prove that this is indeed the case, the inference rule for the annotation of the if-statement is used as a starting point. This rule establishes that at point $B2$, $\{L1 \vee L2\}$ holds. As is shown below, both $L1$ and $L2$ imply $B2$, which is enough to conclude the proof.

First is shown, by a series of implications, that $L1 \Rightarrow B2$. For this, a second property of the $List$ predicate, described in 12.3, is relied on. This property is:

$$LProp2 \quad : \quad List.\lambda.E^{Obj}.a_1.a_2.c \Rightarrow (E^{Obj} = \textbf{nil} \Rightarrow \lambda = \varepsilon)$$

While fairly detailed, some of the more obvious implication steps have been omitted for the sake of brevity and readability:

$$
\begin{array}{ll}
 & L1 \\
\Leftrightarrow & \{\text{def } L1\} \\
 & \textbf{result} = x \wedge z = \textbf{nil} \wedge z = D_1 \wedge x = D_0 \wedge Pre_{Node.LSum} \\
\Leftrightarrow & \{\text{Substitutions } z = D_1 \text{ and } x = D_0\} \\
 & \textbf{result} = D_0 \wedge z = \textbf{nil} \wedge \textbf{nil} = D_1 \wedge x = D_0 \wedge Pre_{Node.LSum} \\
\Rightarrow & \{(P \wedge z = \textbf{nil} \wedge x = D_0) \Rightarrow P\} \\
 & \textbf{result} = D_0 \wedge \textbf{nil} = D_1 \wedge Pre_{Node.LSum} \\
\Leftrightarrow & \{\text{def } Pre_{Node.LSum}\} \\
 & \textbf{result} = D_0 \wedge \textbf{nil} = D_1 \wedge (\textbf{this} \mapsto [\text{val: } D_0, \text{ next: } D_1]^{Node} * NList.L.D_1) \\
\Leftrightarrow & \{LProp2 \text{ using } NList.L.D_1 \text{ and } \textbf{nil} = D_1\} \\
 & L = \varepsilon \wedge \textbf{result} = D_0 \wedge \textbf{nil} = D_1 \wedge (\textbf{this} \mapsto [\text{val: } D_0, \text{ next: } D_1]^{Node} * NList.L.D_0) \\
\Leftrightarrow & \{\text{def } Pre_{Node.LSum}\} \\
 & L = \varepsilon \wedge \textbf{result} = D_0 \wedge \textbf{nil} = D_1 \wedge Pre_{Node.LSum} \\
\Rightarrow & \{(P \wedge \textbf{nil} = D_1) \Rightarrow P\} \\
 & L = \varepsilon \wedge \textbf{result} = D_0 \wedge Pre_{Node.LSum} \\
\Leftrightarrow & \{\text{def } Sum\} \\
 & L = \varepsilon \wedge \textbf{result} = Sum.[D_0] \wedge Pre_{Node.LSum} \\
\Leftrightarrow & \{\lambda = (\lambda +\!\!+ \varepsilon)\} \\
 & L = \varepsilon \wedge \textbf{result} = Sum.([D_0] +\!\!+ \varepsilon) \wedge Pre_{Node.LSum} \\
\Leftrightarrow & \{\text{Substitution } L = \varepsilon\} \\
 & L = \varepsilon \wedge \textbf{result} = Sum.([D_0] +\!\!+ L) \wedge Pre_{Node.LSum} \\
\Rightarrow & \{(P \wedge L = \varepsilon) \Rightarrow P\} \\
 & \textbf{result} = Sum.([D_0] +\!\!+ L) \wedge Pre_{Node.LSum} \\
\Leftrightarrow & \{\text{def } Post_{Node.LSum}\} \\
 & Post_{Node.LSum}
\end{array}
$$

What remains to be shown is that $L2 \Rightarrow B2$. Again a series of implications is presented:

$$
\begin{array}{ll}
 & L2 \\
\Leftrightarrow & \{\text{def } L2\} \\
 & \mathbf{result} = x + y \wedge B1 \\
\Leftrightarrow & \{\text{def } B1\} \\
 & \mathbf{result} = x + y \wedge (\exists d_0, d_1, l_0.\ ((z \mapsto [\text{val: } d_0, \text{ next: } d_1]^{Node} * NList.l_0.d_1) \wedge y = Sum.(d_0 +\!\!+ l_0)) * \\
 & (\mathbf{this} \mapsto [\text{val: } D_0, \text{ next: } D_1]^{Node} \wedge L = ([d_0] +\!\!+ l_0) \wedge z = D_1 \wedge x = D_0)) \\
\Leftrightarrow & \\
 & \exists d_0, d_1, l_0.\ (\mathbf{this} \mapsto [\text{val: } D_0, \text{ next: } D_1]^{Node} * z \mapsto [\text{val: } d_0, \text{ next: } d_1]^{Node} * NList.l_0.d_1) \wedge \\
 & y = Sum.(d_0 +\!\!+ l_0) \wedge L = ([d_0] +\!\!+ l_0) \wedge z = D_1 \wedge x = D_0 \wedge \mathbf{result} = x + y \\
\Leftrightarrow & \{\text{Substitutions } x = D_0,\ y = Sum.(d_0 +\!\!+ l_0) \text{ and } z = D_1\} \\
 & \exists d_0, d_1, l_0.\ (\mathbf{this} \mapsto [\text{val: } D_0, \text{ next: } D_1]^{Node} * D_1 \mapsto [\text{val: } d_0, \text{ next: } d_1]^{Node} * NList.l_0.d_1) \wedge \\
 & y = Sum.(d_0 +\!\!+ l_0) \wedge L = ([d_0] +\!\!+ l_0) \wedge z = D_1 \wedge x = D_0 \wedge \mathbf{result} = D_0 + Sum.(d_0 +\!\!+ l_0) \\
\Rightarrow & \{(P \wedge x = D_0 \wedge y = Sum.(d_0 +\!\!+ l_0) \wedge z = D_1) \Rightarrow P\} \\
 & \exists d_0, d_1, l_0.\ (\mathbf{this} \mapsto [\text{val: } D_0, \text{ next: } D_1]^{Node} * D_1 \mapsto [\text{val: } d_0, \text{ next: } d_1]^{Node} * NList.l_0.d_1) \wedge \\
 & L = ([d_0] +\!\!+ l_0) \wedge \mathbf{result} = D_0 + Sum.(d_0 +\!\!+ l_0) \\
\Leftrightarrow & \{\text{def } NList\} \\
 & \exists d_0, d_1, l_0.\ (\mathbf{this} \mapsto [\text{val: } D_0, \text{ next: } D_1]^{Node} * NList.([d_0] +\!\!+ l_0).D_1) \wedge \\
 & L = ([d_0] +\!\!+ l_0) \wedge \mathbf{result} = D_0 + Sum.(d_0 +\!\!+ l_0) \\
\Leftrightarrow & \{\text{Substitution } L = ([d_0] +\!\!+ l_0)\} \\
 & \exists d_0, d_1, l_0.\ (\mathbf{this} \mapsto [\text{val: } D_0, \text{ next: } D_1]^{Node} * NList.L.D_1) \wedge \\
 & L = ([d_0] +\!\!+ l_0) \wedge \mathbf{result} = D_0 + Sum.L \\
\Rightarrow & \{(P \wedge L = ([d_0] +\!\!+ l_0)) \Rightarrow P;\ \text{def } Pre_{Node.LSum}\} \\
 & \exists d_0, d_1, l_0.\ Pre_{Node.LSum} \wedge \mathbf{result} = D_0 + Sum.L \\
\Leftrightarrow & \{\text{def } Sum\} \\
 & \exists d_0, d_1, l_0.\ Pre_{Node.LSum} \wedge \mathbf{result} = Sum.(D_0 + L) \\
\Leftrightarrow & \{\exists d.P \Leftrightarrow P \text{ when d not free in } P\} \\
 & Pre_{Node.LSum} \wedge \mathbf{result} = Sum.(D_0 + L) \\
\Leftrightarrow & \{\text{def } Post_{Node.LSum}\} \\
 & Post_{Node.LSum}
\end{array}
$$

This concludes the proof and the example.

A.2 RemAll

The second example is of a higher complexity, as it changes the structure of the list. It also uses parameters, an extension presented in 12.1. The specification uses the predicate $NList$ that was defined in the previous example. The method RemAll removes all Nodes in an $NList$ that have value in their val-field equal to the method's parameter. It will, however, retain the list structure. This example also demonstrates how a method can (safely) remove its **this** object.

When the method is called, it first ensures that the part of the list it can reach is cleared of all Nodes with the specified value. Then, if the Node itself needs to be removed, it calls the second method that is specified, RemNext, on the Node that is right above it in the list, and makes sure the list structure is preserved by using the right parameter. For the first Node of a list, a slightly different initialization method is needed, which can than call this method on the second Node. For simplicity, this method is not given here.

Problem Specification
The method is specified by giving the update to the abstract value that the list represents, which is a logical list. To this end, $FreeOf : AExp^t \times LogList \rightarrow LogList$ is defined.

This can be done using structural induction on its list argument:

$$FreeOf.E^t.\varepsilon \stackrel{def}{=} \varepsilon$$

$$FreeOf.E_1^t.([E_2^t] +\!\!+ \lambda) \stackrel{def}{=} \begin{cases} E_1^t = E_2^t & FreeOf.E_1^t.\lambda \\ E_1^t \neq E_2^t & ([E_2^t] +\!\!+ FreeOf.E_1^t.\lambda) \end{cases}$$

It can then be added to the assertion language by extending the definition of the set $LogList$:

$$\lambda ::= ... \mid FreeOf.E^t.\lambda_1$$

The specification of the problem is given below, where
tbr $\in FPar^{Int}$, prev, newnext $\in FPar^{Obj}$, x $\in TempVar^{Obj}$ and y $\in TempVar^{Int}$:

Class Node { val, next;
  Method RemAll (tbr, prev) {
    $< Pre_{Node.RemAll}$: prev $\mapsto$ [val: D_0, next: **this**]^{Node} $*$ **this** $\mapsto$ [val: D_1, next: D_2]^{Node} $*$
    $NList.L.D_2 >$
    ;x := **this**.next
    ;**if** x = nil **then skip**
    else x.RemAll(tbr, **this**)
    fi
    ;y := **this**.val
    ;**if** y = tbr **then**
      x := **this**.next
      ;prev.RemNext(x)
    else skip
    fi
    $< Post_{Node.RemAll}$: $\exists d, l.$ (prev $\mapsto$ [val: D_0, next: d]^{Node} $*$ $NList.l.d$) $\wedge$
    $l = FreeOf$.tbr.$([D_1] +\!\!+ L) >$
  }

  Method RemNext (newnext) {
    $< Pre_{Node.RemNext}$: **this** $\mapsto$ [val: D_0, next: D_1]^{Node} $*$ $D_1 \mapsto []^{Node} >$
    x := **this**.next
    ;**dispose**(x)
    ;**this**.next := newnext
    $< Post_{Node.RemNext}$: **this** $\mapsto$ [val: D_0, next : newnext]$^{Node} >$
}

An Annotation

The specification for the body of the method RemNext is considered trivial and no formal proof is given. Note that this body contains statements that are often the source of many problems. It attests to the power of the proof system that dealing with this body now comes down to an almost direct application of the inference rules for their respective statement-specific inference rules.
For the method RemAll, a fully and correctly annotated version is given below. It is followed by a more detailed look at four of the most interesting points of the proof, labeled B1 to B4. To improve readabilty, some blank lines are added to the annotation:

Method RemAll (tbr, prev) {
$< \text{prev} \mapsto [\text{val: } D_0, \text{ next: } \mathbf{this}]^{Node} * \mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } D_2]^{Node} * NList.L.D_2 >$
;x := **this**.next
$< x = D_2 \wedge Pre_{Node.RemAll} >$
;**if** x = nil **then**
$< x = nil \wedge x = D_2 \wedge Pre_{Node.RemAll} >$
skip
$< L1 : x = nil \wedge x = D_2 \wedge Pre_{Node.RemAll} >$
else

$< x \neq nil \wedge x = D_2 \wedge Pre_{Node.RemAll} >$
x.RemAll(tbr, **this**)
$< B1 : (\exists d_0, l_0, d, l.\ (\mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } d]^{Node} * NList.l.d) \wedge l = FreeOf.\text{tbr}.L) *$
$(\text{prev} \mapsto [\text{val: } D_0, \text{ next: } \mathbf{this}]^{Node} \wedge L = ([d_0] +\!\!+ l_0) \wedge x = D_2) >$

fi
$< B2 : \exists d, l.\ (\text{prev} \mapsto [\text{val: } D_0, \text{ next: } \mathbf{this}]^{Node} * \mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } d]^{Node} * NList.l.d) \wedge$
$l = FreeOf.\text{tbr}.L >$
;y := **this**.val
$< y = D_1 \wedge B2 >$
;**if** y = tbr **then**
$< y = \text{tbr} \wedge y = D_1 \wedge B2 >$
x := **this**.next

$< \exists d, l.\ x = d \wedge y = \text{tbr} \wedge y = D_1 \wedge (\text{prev} \mapsto [\text{val: } D_0, \text{ next: } \mathbf{this}]^{Node} *$
this $\mapsto [\text{val: } D_1, \text{ next: } d]^{Node} * NList.l.d) \wedge l = FreeOf.\text{tbr}.L >$
;prev.RemNext(x)
$< B3 : \exists d, l.\ \text{prev} \mapsto [\text{val: } D_0, \text{ next: } x]^{Node} *$
$(NList.l.d \wedge l = FreeOf.\text{tbr}.L \wedge x = d \wedge y = \text{tbr} \wedge y = D_1) >$

else
$< y \neq \text{tbr} \wedge y = D_1 \wedge B2 >$
skip
$< L2 : y \neq \text{tbr} \wedge y = D_1 \wedge B2 >$
fi
$< B4 : \exists d, l.\ (\text{prev} \mapsto [\text{val: } D_0, \text{ next: } d]^{Node} * NList.l.d) \wedge l = FreeOf.\text{tbr}.([D_1] +\!\!+ L) >$
}

The Recursive Call B1
The first non-trivial specification that is encountered is the specification around the statement x.RemAll(tbr, **this**). A derivation tree is given below, followed by an explanation. In this derivation tree the label P1 stands for:

$P1 \quad : \quad \text{prev} \mapsto [\text{val: } D_0, \text{ next: } \mathbf{this}]^{Node} \wedge L = ([d_0] +\!\!+ l_0) \wedge x = D_2$

$$\dfrac{\dfrac{}{Pre_{Node.RemAll}[x/\mathbf{this}, \mathbf{this}/\mathrm{prev}] \Rightarrow (x \mapsto []^{Node} * \mathbf{true})}\,A}{\begin{array}{c}\{Pre_{Node.RemAll}[x/\mathbf{this}, \mathbf{this}/\mathrm{prev}]\}\\ \text{x.RemAll(tbr, } \mathbf{this})\\ \{Post_{Node.RemAll}[x/\mathbf{this}, \mathbf{this}/\mathrm{prev}]\}\end{array}}\,Pre, Post$$

$$\dfrac{\begin{array}{c}\{(\mathbf{this} \mapsto [\text{val: } D_0, \text{ next: } x]^{Node} * x \mapsto [val : D_1, next : D_2]^{Node} * NList.L.D_2)\}\\ \text{x.RemAll(tbr, } \mathbf{this})\\ \{\exists d, l.\ (\text{ this} \mapsto [\text{val: } D_0, \text{ next: } d]^{Node} * NList.l.d) \wedge l = FreeOf.\text{tbr.}([D_1] +\!\!+ L)\}\end{array}}{\begin{array}{c}\{(\mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } x]^{Node} * x \mapsto [val : d_0, next : d_1]^{Node} * NList.L.d_1)\}\\ \text{x.RemAll(tbr, } \mathbf{this})\\ \{\exists d, l.\ (\mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } d]^{Node} * NList.l.d) \wedge l = FreeOf.\text{tbr.}([d_0] +\!\!+ L)\}\end{array}}\,D\text{-elim, 3x}$$

$$\dfrac{\begin{array}{c}\{(\mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } x]^{Node} * x \mapsto [val : d_0, next : d_1]^{Node} * NList.l_0.d_1)\}\\ \text{x.RemAll(tbr, } \mathbf{this})\\ \{\exists d, l.\ (\mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } d]^{Node} * NList.l.d) \wedge l = FreeOf.\text{tbr.}([d_0] +\!\!+ l_0)\}\end{array}}{\begin{array}{c}\{(\mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } x]^{Node} * x \mapsto [val : d_0, next : d_1]^{Node} * NList.l_0.d_1) * P1\}\\ \text{x.RemAll(tbr, } \mathbf{this})\\ \{(\exists d, l.\ (\mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } d]^{Node} * NList.l.d) \wedge l = FreeOf.\text{tbr.}([d_0] +\!\!+ l_0)) * P1\}\end{array}}\,L\text{-elim}$$

$$\dfrac{\begin{array}{c}\{(\mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } x]^{Node} * x \mapsto [val : d_0, next : d_1]^{Node} * NList.l_0.d_1) * P1\}\\ \text{x.RemAll(tbr, } \mathbf{this})\\ \{(\exists d, l.\ (\mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } d]^{Node} * NList.l.d) \wedge l = FreeOf.\text{tbr.}([d_0] +\!\!+ l_0)) * P1\}\end{array}}{\begin{array}{c}\{\exists d_0, d_1, l_0.\ (\mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } x]^{Node} * x \mapsto [val : d_0, next : d_1]^{Node} * NList.l_0.d_1) * P1\}\\ \text{x.RemAll(tbr, } \mathbf{this})\\ \{\exists d_0, d_1, l_0, d, l.\ ((\mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } d]^{Node} * NList.l.d) \wedge l = FreeOf.\text{tbr.}([d_0] +\!\!+ l_0)) * P1)\}\end{array}}\,FR$$

$$\dfrac{\begin{array}{c}\{\exists d_0, d_1, l_0.\ (\mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } x]^{Node} * x \mapsto [val : d_0, next : d_1]^{Node} * NList.l_0.d_1) * P1\}\\ \text{x.RemAll(tbr, } \mathbf{this})\\ \{\exists d_0, d_1, l_0, d, l.\ ((\mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } d]^{Node} * NList.l.d) \wedge l = FreeOf.\text{tbr.}([d_0] +\!\!+ l_0)) * P1)\}\end{array}}{\begin{array}{c}\{x \neq \mathbf{nil} \wedge x = D_2 \wedge Pre_{Node.RemAll}\}\\ \text{x.RemAll(tbr, } \mathbf{this})\\ \{\exists d_0, l_0, d, l.\ ((\mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } d]^{Node} * NList.l.d) \wedge l = FreeOf.\text{tbr.}L) * P1)\}\end{array}}\,ROC$$

As in the previous example, the first deduction in the tree is an application of the Rule of Consequence. This time, both a strengthening of the precondition and a weakening of the postcondition are required. Most complicated is the implication used for the strengthening of the precondition. Again like in the previous example, the proof relies on a property of the *List* predicate:

$$LProp1 \quad : \quad List.\lambda.E^{Obj}.a_1.a_2.c \Rightarrow (E^{Obj} \neq \mathbf{nil} \Rightarrow (\exists d, l_0.\ \lambda = [d] +\!\!+ l_0))$$

The relevant implication is presented by a series of implications leading from the precondition to the weaker precondition. Note that this is not a full deduction, several of the more obvious steps have been omitted. Blank lines have been added to the series to improve readability only:

(1) $x \neq \mathbf{nil} \wedge x = D_2 \wedge Pre_{Node.RemAll}$

$\Leftrightarrow$ {def $Pre_{Node.RemAll}$}
$x \neq \mathbf{nil} \wedge x = D_2 \wedge (\text{prev} \mapsto [\text{val: } D_0, \text{ next: } \mathbf{this}]^{Node} * \mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } D_2]^{Node} * NList.L.D_2)$

$\Leftrightarrow$ {Substitution $x = D_2$}
$x \neq \mathbf{nil} \wedge x = D_2 \wedge (\text{prev} \mapsto [\text{val: } D_0, \text{ next: } \mathbf{this}]^{Node} * \mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } x]^{Node} * NList.L.x)$

$\Leftrightarrow$ {$LProp1$, using $NList.L.x$ and $x \neq \mathbf{nil}$}
$\exists d_0, l_0.\ L = ([d_0] +\!\!+ l_0) \wedge x \neq \mathbf{nil} \wedge x = D_2 \wedge (\text{prev} \mapsto [\text{val: } D_0, \text{ next: } \mathbf{this}]^{Node} *$
$\mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } x]^{Node} * NList.L.x)$

$\Rightarrow$ {$(P \wedge x \neq \mathbf{nil}) \Rightarrow P$}
$\exists d_0, l_0.\ L = ([d_0] +\!\!+ l_0) \wedge x = D_2 \wedge (\text{prev} \mapsto [\text{val: } D_0, \text{ next: } \mathbf{this}]^{Node} *$
$\mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } x]^{Node} * NList.L.x)$

$\Leftrightarrow$ {Substitution $L = ([d_0] +\!\!+ l_0)$; def $NList$}
$\exists d_0, l_0.\ L = ([d_0] +\!\!+ l_0) \wedge x = D_2 \wedge (\text{prev} \mapsto [\text{val: } D_0, \text{ next: } \mathbf{this}]^{Node} *$
$\mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } x]^{Node} * (\exists d_1.\ x \mapsto [val : d_0, next : d_1]^{Node} * NList.l_0.d_1))$

$\Leftrightarrow$
$\exists d_0, d_1, l_0.\ (\mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } x]^{Node} * x \mapsto [val : d_0, next : d_1]^{Node} * NList.l_0.d_1) *$
$(\ \text{prev} \mapsto [\text{val: } D_0, \text{ next: } \mathbf{this}]^{Node} \wedge L = ([d_0] +\!\!+ l_0) \wedge x = D_2)$

$\Leftrightarrow$ {def $P1$}
$\exists d_0, d_1, l_0.\ (\mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } x]^{Node} * x \mapsto [val : d_0, next : d_1]^{Node} * NList.l_0.d_1) * P1$

The implication required for the weakening of the postcondition can be shown to hold by the elimination of d_1 which does not occur in the final postcondition and the substitution of $([d_0 +\!\!+ l_0])$ by L which is allowed because this occurs as an equality in $P1$.

The next step is the application of the rule for Auxiliary Variable introduction (AV), which is preceded by an implicit application of the Rule of Consequence for the postcondition, where d and l do not occur in P_1.

Then, the Frame Rule (FR) can be applied, which is allowed as the method call does not modify any variables.

L-elimination and D-elimination are used for the following substitutions: $l_0/L, D_1/D_0, d_0/D_1$ and d_1/D_2.

This produces a shape that allows for the replacement with $Pre_{Node.RemAll}$ and $Post_{Node.RemAll}$.

Now, following the same line of reasoning as in the previous example, the assumption specified by the rule for method call is applied which only leaves the proof of the validity of $Pre_{Node.RemAll}[x/\mathbf{this}, \mathbf{this}/\text{prev}] \Rightarrow (x \mapsto []^{Node} * \mathbf{true})$. This proof is similar to the one presented before and will not be given here.

That concludes the treatment of this recursive call specification.

If-statement B2
The annotation rule for the if-statement presented in 10.3 shows that after the statement $L1 \vee B1$. When both can be shown to imply B, the Rule of Consequence can be applied to achieve the proof of the validity of the specification.

The implication $L1 \Rightarrow B2$ relies on a property of the *List* predicate repeated below:

$$LProp2 \quad : \quad List.\lambda.E^{Obj}.a_1.a_2.c \Rightarrow (E^{Obj} = \textbf{nil} \Rightarrow \lambda = \varepsilon)$$

The implication is shown to hold by the following series of implications. Again, several steps have been omitted for the sake of readability and brevity:

$L1$
$\Leftrightarrow$ {def L_1}
$x = nil \wedge x = D_2 \wedge Pre_{Node.RemAll}$
$\Leftrightarrow$ {Substitution $x = D_2$}
$D_2 = nil \wedge x = D_2 \wedge Pre_{Node.RemAll}$
$\Rightarrow$ {$(P \wedge x = D_2) \Rightarrow P$}
$D_2 = nil \wedge Pre_{Node.RemAll}$
$\Leftrightarrow$ {def $Pre_{Node.RemAll}$}
$D_2 = nil \wedge (\text{prev} \mapsto [\text{val: } D_0, \text{ next: } \textbf{this}]^{Node} * \textbf{this} \mapsto [\text{val: } D_1, \text{ next: } D_2]^{Node} * NList.L.D_2)$
$\Leftrightarrow$ {$LProp2$, using $NList.L.D_2$ and $D_2 = nil$}
$L = \varepsilon \wedge D_2 = nil \wedge (\text{prev} \mapsto [\text{val: } D_0, \text{ next: } \textbf{this}]^{Node} * \textbf{this} \mapsto [\text{val: } D_1, \text{ next: } D_2]^{Node} * NList.L.D_2)$
$\Leftrightarrow$ {Substitution $L = \varepsilon$}
$L = \varepsilon \wedge D_2 = nil \wedge (\text{prev} \mapsto [\text{val: } D_0, \text{ next: } \textbf{this}]^{Node} * \textbf{this} \mapsto [\text{val: } D_1, \text{ next: } D_2]^{Node} * NList.\varepsilon.D_2)$
$\Rightarrow$ {def $FreeOf$; Substitution rules; Weakening}
$\varepsilon = FreeOf.\text{tbr}.L \wedge (\text{prev} \mapsto [\text{val: } D_0, \text{ next: } \textbf{this}]^{Node} * \textbf{this} \mapsto [\text{val: } D_1, \text{ next: } D_2]^{Node} * NList.\varepsilon.D_2)$
$\Rightarrow$ {$P \Rightarrow \exists l.\ P[l/\varepsilon]$}
$\exists l.\ l = FreeOf.\text{tbr}.L \wedge (\text{prev} \mapsto [\text{val: } D_0, \text{ next: } \textbf{this}]^{Node} * \textbf{this} \mapsto [\text{val: } D_1, \text{ next: } D_2]^{Node} * NList.l.D_2)$
$\Rightarrow$ {$P \Rightarrow \exists d.\ P[d/D_2]$}
$\exists d, l.\ l = FreeOf.\text{tbr}.L \wedge (\text{prev} \mapsto [\text{val: } D_0, \text{ next: } \textbf{this}]^{Node} * \textbf{this} \mapsto [\text{val: } D_1, \text{ next: } d]^{Node} * NList.l.d)$
$\Leftrightarrow$ {def $B2$}
$B2$

Proof of $B1 \Rightarrow B2$ is simpler. An argumentation is presented below:

$B1$
$\Leftrightarrow$ {def $B1$}
$(\exists d, l.\ (\textbf{this} \mapsto [\text{val: } D_1, \text{ next: } d]^{Node} * NList.l.d) \wedge l = FreeOf.\text{tbr}.L) *$
$(\text{prev} \mapsto [\text{val: } D_0, \text{ next: } \textbf{this}]^{Node} \wedge L = ([d_0] +\!\!+ l_0) \wedge x = D_2)$
$\Rightarrow$ {$(P \wedge L = ([d_0] +\!\!+ l_0) \wedge x = D_2) \Rightarrow P$}
$(\exists d, l.\ (\textbf{this} \mapsto [\text{val: } D_1, \text{ next: } d]^{Node} * NList.l.d) \wedge l = FreeOf.\text{tbr}.L) *$
$\text{prev} \mapsto [\text{val: } D_0, \text{ next: } \textbf{this}]^{Node}$
$\Leftrightarrow$ {def B2}
$B2$

This concludes the proof of B2

Method Call RemNext, B3
Proof of the specification around the call of method RemNext is fairly straightforward. In the derivation tree presented below, the label $P1$ is used for:

$$P1 \quad : \quad NList.l.d \wedge l = FreeOf.\text{tbr}.L \wedge x = d \wedge y = \text{tbr} \wedge y = D_1$$

The derivation assumes the existence of a proof of the specification of the body of the method, which it will refer to as $PofB$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{Pre_{Node.RemNext}[\text{prev}/\mathbf{this}, x/\text{newnext}] \Rightarrow \text{prev} \mapsto []^{Node} * \mathbf{true}} \quad PofB}{\{Pre_{Node.RemNext}[\text{prev}/\mathbf{this}, x/\text{newnext}]\}\ \text{prev.RemNext(x)}\ \{Post_{Node.RemNext}[\text{prev}/\mathbf{this}, x/\text{newnext}]\}}\ \text{Method Call}}{\{\text{prev} \mapsto [\text{val: } D_0, \text{ next: } D_1]^{Node} * D_1 \mapsto []^{Node}\}\ \text{prev.RemNext(x)}\ \{\text{prev} \mapsto [\text{val: } D_0, \text{ next: } x]^{Node}\}}\ def\ Pre, Post}{\{\text{prev} \mapsto [\text{val: } D_0, \text{ next: } \mathbf{this}]^{Node} * \mathbf{this} \mapsto []^{Node}\}\ \text{prev.RemNext(x)}\ \{\text{prev} \mapsto [\text{val: } D_0, \text{ next: } x]^{Node}\}}\ D\text{-elim}}{\{\text{prev} \mapsto [\text{val: } D_0, \text{ next: } \mathbf{this}]^{Node} * \mathbf{this} \mapsto []^{Node} * P1\}\ \text{prev.RemNext(x)}\ \{\text{prev} \mapsto [\text{val: } D_0, \text{ next: } x]^{Node} * P1\}}\ \text{Frame Rule}}{\{\exists d, l.\ \text{prev} \mapsto [\text{val: } D_0, \text{ next: } \mathbf{this}]^{Node} * \mathbf{this} \mapsto []^{Node} * P1\}\ \text{prev.RemNext(x)}\ \{\exists d, l.\ \text{prev} \mapsto [\text{val: } D_0, \text{ next: } x]^{Node} * P1\}}\ \text{AV-intro}}{\{\exists d, l.\ x = d \wedge y = \text{tbr} \wedge y = D_1 \wedge (\text{prev} \mapsto [\text{val: } D_0, \text{ next: } \mathbf{this}]^{Node} * \mathbf{this} \mapsto [\text{val: } D_1, \text{ next: } d]^{Node} * NList.l.d) \wedge l = FreeOf.\text{tbr}.L\}\ \text{prev.RemNext(x)}\ \{\exists d, l.\ \text{prev} \mapsto [\text{val: } D_0, \text{ next: } x]^{Node} * P1\}}\ ROC$$

This method call demonstrates how a local specification can lead to a fairly simple proof. For instance, no knowledge of the newnext parameter is required as this is of no importance to the safety of this method.

If-statement B4
In this case, the proof obligation is $B3 \vee L2 \Rightarrow B4$

To see why $B3 \Rightarrow B4$, note that d can be substituted for x, and that
$FreeOf.\text{tbr}.L = FreeOf.\text{tbr}.([tbr] +\!\!+ L) = FreeOf.\text{tbr}.([y] +\!\!+ L) = FreeOf.\text{tbr}.([D_1] +\!\!+ L)$.

The proof of $L2 \Rightarrow B4$ is harder and requires the use of the definition of $NList$. As similar proofs have been encountered earlier, a shorth argumentation will be given instead.
As *prev* points to **this** and **this** points to an $Nlist$ the combination is the implementation of a list structure. As $D_1 \neq \text{tbr}$ this list structure will represent the logical list $FreeOf.\text{tbr}.([D_1] +\!\!+ L)$.

That concludes the treatment of B4, and with that the proof of this example.

B Soundness

This chapter presents the proofs of the soundness of the command-specific rules for program annotation presented in chapter 10.
To avoid the use of double subscripts in the proofs, $g\sigma_i$ is written instead of g_{σ_i}. For the same reason $|[E]|\sigma_i$ is written instead of $|[E]|_{\sigma_i}$.

The form $L1 : \sigma \models P$ is used to label some of the conclusions, so they can be referred to later on.

B.1 Simple Assignment

This section proves soundness for the inference rule for Simple Assignment. This axiom has the following shape:

Simple Assignment:
$\{P\}\ x^t := e^t\ \{\exists d.\ x^t = e^t[d/x^t] \wedge P[d/x^t]\}$

Soundness is proved in the following way:

Assume without loss of generality that $FV_D(P) = \{D_1, \ldots, D_i\}$.
Furthermore, assume a τ and $\delta_1, \ldots, \delta_i$ such that:

$\tau' \models P$, where $\tau' = (\tau \mid u_\tau(\forall j.\ 1 \leq j \leq i.\ D_j \rightarrow \delta_j))$

The semantics of specifications now imply that the proof obligation for such a τ is:

$< x^t := e^t, (s_\tau, f_\tau) >$ is safe and
$\forall (s, f) \in OpState:\ (< x^t := e^t, (s_\tau, f_\tau) > \rightsquigarrow\ (s, f)$ implies
$(\tau' \mid s_{\tau'} \rightarrow s, f_{\tau'} \rightarrow f) \models \exists d.\ x = e^t[d/x^t] \wedge P[d/x^t])$

The operational semantics 4 define only a single transition rule for the configuration $< x^t := e^t, (s_\tau, f_\tau) >$. As this rule does not lead to **abort** as a terminal state, this configuration is safe.

Now assume the premise of this rule to hold (i.e. $|[e^t]|s_\tau = \delta$ for some δ). To prove is then:

$(\tau' \mid s_{\tau'}(x^t \rightarrow \delta)) \models \exists d.\ x = e^t[d/x^t] \wedge P[d/x^t]$

Assume d not in P (without loss of generality). Then the truth value of P does not depend on the mapping of d. $\tau' \models P$ holds by assumption. Therefore the following must hold:

$(\tau' \mid q_{\tau'}(d \rightarrow |[x^t]|_{\tau'})) \models P$

The next implication relies on the following lemma:

Lemma B.1 ($\sigma \models P$ and $|[E_1]|_\sigma = |[E_2]|_\sigma$) implies $\sigma \models P[E_1/E_2]$

This lemma can be proven using structural induction on P.

As in the state above the valuations of d and x are equal by construction, the lemma can be applied:

$(\tau' \mid q_{\tau'}(d \rightarrow |[x^t]|_{\tau'})) \models P[d/x^t]$

As x^t does not occur in $P[d/x^t]$, it's value can be remapped:

$$SA1 \quad : \quad (\tau' \mid q_{\tau'}(d \rightarrow |[x^t]|_{\tau'}), s_{tau'} \rightarrow \delta) \models P[d/x^t]$$

Now assume, without loss of generality, that D does not occur in P. That means:

$$(\tau' \mid u_{\tau'}(D \rightarrow \delta)) \models P$$

In this state valuations of D and e^t both produce δ. The semantics of $=$ then show:

$$(\tau' \mid u_{\tau'}(D \rightarrow \delta)) \models D = e^t$$

As above, substitute d for x^t, and then use general rules of substitution to arrive at:

$$(\tau' \mid q_{\tau'}(d \rightarrow |[x^t]|_{\tau'}), u_{\tau'}(D \rightarrow \delta)) \models D = e^t[d/x^t]$$

As x^t does not occur in $D = e^t[d/x^t]$ its value can be remapped without changing the truth value of the proposition. Map it to δ and use the fact that D and x^t then both valuate to δ to apply lemma B.1:

$$(\tau' \mid s_{\tau'}(x^t \rightarrow \delta), q_{\tau'}(d \rightarrow |[x^t]|_{\tau'}), u_{\tau'}(D \rightarrow \delta)) \models (D = e^t[d/x^t])[x^t/D]$$

Again relying on general rules of substitution, and the fact that D does not occur in $e^t[d/x^t]$:

$$(\tau' \mid s_{\tau'}(x^t \rightarrow \delta), q_{\tau'}(d \rightarrow |[x^t]|_{\tau'}), u_{\tau'}(D \rightarrow \delta)) \models x^t = e^t[d/x^t]$$

Now D no longer occurs in the proposition and its mapping can safely be omitted. Therefore:

$$(\tau' \mid s_{\tau'}(x^t \rightarrow \delta), q_{\tau'}(d \rightarrow |[x^t]|_{\tau'})) \models x^t = e^t[d/x^t]$$

Combining the conclusion above with that of $SA1$ the semantics of $\wedge$ can now be used to conclude:

$$(\tau' \mid s_{\tau'}(x^t \rightarrow \delta), q_{\tau'}(d \rightarrow |[x^t]|_{\tau'})) \models x^t \doteq e^t[d/x^t] \wedge P[d/x^t]$$

The final step relies on the semantics of $\exists$:

$$(\tau' \mid s_{\tau'}(x^t \rightarrow \delta)) \models \exists d.\ x^t \doteq e^t[d/x^t] * P[d/x^t]$$

This concludes the proof and proves the soundness of the axiom.

B.2 Object Component Assignment

The axiom that is the inference rule for Object Component Assignment is proven sound in this section. As the rule uses a direct application of the frame rule, already proven sound in [9], only a local version of this rule has to be proven. This rule is:

Object Component Assignment, local:
$\{o \mapsto [a^t : E^t, fields_n]^c\}\ o.a^t := e^t\ \{o \mapsto [a^t : e^t, fields_n]^c\}$

Soundness can be proved in the following way:
Assume without loss of generality that $FV_D(o \mapsto [a^t : E^t, fields_n]^c) = \{D_1, \ldots, D_i\}$.
Furthermore, assume a τ and $\delta_1, \ldots, \delta_i$ such that:

$$\tau' \models o \mapsto [a^t : E^t, fields_n]^c, \text{ where } \tau' = (\tau \mid u_\tau(\forall j.\ 1 \leq j \leq i.\ D_j \rightarrow \delta_j))$$

The semantics of specifications now imply that the proof obligation for such an τ is:

$< o.a^t := e^t, (s_\tau, f_\tau) >$ is safe and
$\forall (s, f) \in OpState:\ (< o.a^t := e^t, (s_\tau, f_\tau) > \rightsquigarrow\ (s, f)$ implies
$(\tau' \mid s_{\tau'} \rightarrow s, f_{\tau'} \rightarrow f) \models o \mapsto [a^t : e^t, fields_n]^c)$

The operational semantics (chapter 4) specify two possible transition rules for a configuration $< o.a^t := e^t, (s_\sigma, f_\sigma) >$. One of these rules could endanger the safety demand above by leading to **abort**. The premise of that rule is: $o \notin dom(s)$ *or* $s(o) \notin dom(f)$ *or* $a^t \notin dom(f(s(o)))$. However, the semantics of assertions (chapter 7) ensure that, for $o \mapsto [a^t : E^t, fields_n]^c$ to hold, $dom(f_\tau) = \{|[o]|_\tau\} = \{s_\tau(o)\}$. Additionally, $f_\tau(|[o]|_\tau, a_t) = |[E^t]|_\tau$, which means that $a^t \in dom(f_\tau(s_\tau(o)))$. This proves the safety of $< o.m, (s_\tau, f_\tau) >$.

The configuration can only reach a terminal state by an application of the other operational rule. We assume the premises (which require $|[e^t]|s_\tau = \delta$ for some δ), and have to prove:

$$(\tau' \mid f_{\tau'}((s_\tau(o), a^t) \rightarrow \delta)) \models o \mapsto [a^t : e^t, fields_n]^c$$

This state and τ' differ only on the value of $f_{\tau'}(s_\tau(o), a^t)$. As $s_\tau(o) = |[o]|s_\tau$ and $\delta = |[e^t]|s_\tau = |[e^t]|_\tau$, the semantic of $\mapsto$ show this difference is reflected by the update of E^t to e^t, which concludes the proof.

B.3 Object Component Lookup

The inference rule that allows the annotation of $x^t := o.a^t$ is the following axiom:

Object Component Lookup:
$$\frac{\{o \mapsto [a^t : E^t, fields_n]^c * P\}\ x^t := o.a^t}{\{\exists d.\ x^t = E^t[d/x^t] \wedge (o \mapsto [a^t : E^t]^c * P)[d/x^t]\}}$$

A direct application of the frame rule to the local rule is not a possibility here, as x^t might occur in P but is modified by $x^t := o.a^t$. The solution is to use an existentially quantified variable to take x^t's place.

Soundness can be proved as follows:
Assume without loss of generality that $FV_D(o \mapsto [a^t : E, fields_n]^c) = \{D_1, \ldots, D_i\}$.
Furthermore, assume a τ and $\delta_1, \ldots, \delta_i$ such that:

$\tau' \models o \mapsto [a^t : E^t, fields_n]^c * P$, where $\tau' = (\tau \mid u_\tau(\forall j.\ 1 \leq j \leq i.\ D_j \rightarrow \delta_j))$

The semantics of specifications now imply that the proof obligation for such an τ is:

$< x^t := o.a^t, (s_\tau, f_\tau) >$ is safe and
$\forall (s, f) \in OpState:\ (< x^t := o.a^t, (s_\tau, f_\tau) > \rightsquigarrow\ (s, f)$ implies
$(\tau' \mid s_{\tau'} \rightarrow s, f_{\tau'} \rightarrow f) \models \exists d.\ x^t = E^t[d/x^t] \wedge (o \mapsto [a^t : E^t]^c * P)[d/x^t])$

The operational semantics (chapter 4) specify two possible transition rules for a configuration $< x^t := o.a^t, (s_\sigma, f_\sigma) >$. One of these rules could endanger the safety demand above by leading to **abort**. As the premise of this rule is the same as that was encountered in B.2 the same line of reasoning can be used to show that $< x^t := o.a^t, (s_\tau, f_\tau) >$ is safe.

In fact, this proves that the premise of the only other applicable rule holds.
This rule establishes $< x^t := o.a^t, (s_\tau, f_\tau) > \rightsquigarrow\ ((s_\tau \mid x^t \rightarrow f_\tau(s_\tau(o), a^t)), f_\tau)$

The proof obligation is:

$$(\tau' \mid s_{\tau'}(x^t \rightarrow f_\tau(s_\tau(o), a^t))) \models \exists d.\ x^t = E^t[d/x^t] \wedge (o \mapsto [a^t : E^t]^c * P)[d/x^t]$$

To prove this is true, assume d does not occur in $o \mapsto [a^t : E^t, fields_n]^c * P$ (which can be done without loss of generality). That means the following holds:

$$OCL1 \quad : \quad (\tau' \mid q_{\tau'}(d \rightarrow |[x^t]|_{\tau'})) \models o \mapsto [a^t : E^t, fields_n]^c * P$$

The proof is split into two parts. The first part begins with an implication that relies on lemma B.1: As in the state above d and x^t valuate to the same value, it can be seen that:

$$(\tau' \mid q_{\tau'}(d \rightarrow |[x^t]|_{\tau'})) \models (o \mapsto [a^t : E^t, fields_n]^c * P)[d/x^t]$$

As x^t does not occur in this proposition, its value can safely be changed without changing the truth value of the proposition in the updated state. That means the following holds:

$$OCL2 \quad : \quad (\tau' \mid s_{\tau'}(x^t \rightarrow f_\tau(s_\tau(o), a^t)), q_{\tau'}(d \rightarrow |[x^t]|_{\tau'})) \models (o \mapsto [a^t : E^t, fields_n]^c * P)[d/x^t]$$

The second part of the of the proof is started by recalling that

$$(\tau' \mid q_{\tau'}(d \rightarrow |[x^t]|_{\tau'})) \models o \mapsto [a^t : E^t, fields_n]^c * P$$

(and that, as in $OCL1$, d does not occur in the proposition).
Then the semantics of $\mapsto$ imply that $|[E^t]|_{\tau'} = f_{\tau'}(s_\tau(o), a^t)$. Now assume, without loss of generality, that variable D does not occur in E^t. The semantics of $=$ show that:

$$(\tau' \mid q_{\tau'}(d \rightarrow |[x^t]|_{\tau'}), u_\tau(D \rightarrow f_{\tau'}(s_\tau(o), a^t))) \models D = E^t$$

And, as this state maps d and x to the same value, also:

$$(\tau' \mid q_{\tau'}(d \rightarrow |[x^t]|_{\tau'}), u_\tau(D \rightarrow f_{\tau'}(s_\tau(o), a^t))) \models (D = E^t)[d/x^t]$$

which is equivalent to:

$$(\tau' \mid q_{\tau'}(d \rightarrow |[x^t]|_{\tau'}), u_\tau(D \rightarrow f_{\tau'}(s_\tau(o), a^t))) \models D = E^t[d/x^t]$$

As x does not occur in $D = E^t[d/x^t]$ its value can be remapped to that of D in this state. Then lemma B.1 is be used to arrive at:

$$(\tau' \mid s_{\tau'}(x^t \rightarrow f_\tau(s_\tau(o), a^t)), q_{\tau'}(d \rightarrow |[x^t]|_{\tau'}), u_\tau(D \rightarrow f_{\tau'}(s_\tau(o), a^t))) \models (D = E^t[d/x^t])[x^t/D]$$

As $(D = E^t[d/x^t])[x^t/D]$ is equivalent to $x^t = E^t[d/x^t]$ and D does not occur in $x^t = E^t[d/x^t]$ the following holds:

$$(\tau' \mid s_{\tau'}(x^t \rightarrow f_\tau(s_\tau(o), a^t)), q_{\tau'}(d \rightarrow |[x^t]|_{\tau'})) \models x = E^t[d/x^t]$$

The conclusion above can be combined with $OCL2$ to satisfy the semantics of $\wedge$:

$$(\tau' \mid s_{\tau'}(x^t \rightarrow f_\tau(s_\tau(o), a^t)), q_{\tau'}(d \rightarrow |[x^t]|_{\tau'})) \models x = E^t[d/x^t] \wedge (o \mapsto [a^t : E^t, fields_n]^c * P)[d/x^t]$$

The proof concludes with one more step, relying on the semantics of $\exists$:

$$(\tau' \mid s_{\tau'}(x^t \rightarrow f_\tau(s_\tau(o), a^t))) \models \exists d.\ x = E^t[d/x^t] * (o \mapsto [a^t : E^t, fields_n]^c * P)[d/x^t]$$

B.4 D-Elimination rule

This section proves the soundness of the D-Elimination rule for method call, a variation on the structural variable substition rule from 10.2. The rule to prove is:

D-Elimination rule for method call:

$$\frac{\{P_1\}\ o.m\ \{P_2\}}{\{P_1[E^t/D]\}\ o.m\ \{P_2[E^t/D]\}}$$

We assume that $\{P_1\}\ o.m\ \{P_2\}$ is a valid triple, and prove that $\{P_1[E^t/D]\}\ o.m\ \{P_2[E^t/D]\}$ is then valid as well.

The proof is split into two cases.
Recall that the set of universally quantified variables appearing in a proposition P is given by $FV_D(P)$. As was stated in 10.6, for any valid specification $\{P_1\}\ S\ \{P_2\}$, it can be assumed that $FV_D(P_2) \subseteq FV_D(P_1)$.

In the first case, assume $D \notin FV_D(P_1)$. As $FV_D(P_2) \subseteq FV_D(P_1)$, then also $D \notin FV_D(P_2)$. In that case $P_1[E^t/D] = P_1$ and $P_2[E^t/D] = P_2$, making the rule trivially true.

The second case assumes $D \in FV_D(P_1)$.
Assume, without loss of generality, that $FV_D(P_1[E^t/D]) = \{D_1, \ldots, D_i\}$.
Then the semantics of specifications as described in chapter 9 show that for $\{P_1[E^t/D]\}\ o.m\ \{P_2[E^t/D]\}$ to hold the following must hold:

$$\begin{array}{l}\forall\sigma : (\forall\delta_1, \ldots, \delta_i : (\sigma' \models P_1[E_t/D] \text{ implies} \\ \quad < o.m, (s_\sigma, f_\sigma) > \text{ is safe and} \\ \quad \forall(s,f) \in OpState : (< o.m, (s_\sigma, f_\sigma) > \rightsquigarrow^* \ (s,f) \text{ implies } (\sigma' \mid s_{\sigma'} \rightarrow s, f_{\sigma'} \rightarrow f) \models P_2[E^t/D])))\end{array}$$

where $\sigma' = (\sigma \mid u_\sigma(\forall j.\ 1 \leq j \leq i.\ D_j \rightarrow \delta_j))$

Now assume a σ_1 and $\delta_1, \ldots, \delta_i$ such that

$\sigma_1' \models P_1[E^t/D]$
where $\sigma_1' = (\sigma_1 \mid u\sigma_1(\forall j.\ 0 \leq j \leq i.\ D_j \rightarrow \delta_j))$

The proof obligation is then that

$< o.m, (s\sigma_1, f\sigma_1) >$ is safe and
$\forall(s,f) \in OpState : (< o.m, (s\sigma_1, f\sigma_1) > \rightsquigarrow \ (s,f) \text{ implies } (\sigma_1' \mid s\sigma_1' \rightarrow s, f\sigma_1' \rightarrow f) \models P_2[E^t/D])$

We have assumed the validity of $\{P_1\}\ o.m\ \{P_2\}$. We assume without loss of generality that $FV_D(P_1) = \{D\}$. That means that

$$\begin{array}{l}\forall\sigma : (\forall\delta : (\sigma' \models P_1 \text{ implies} \\ \quad < o.m, (s_\sigma, f_\sigma) > \text{ is safe and} \\ \quad \forall(s,f) \in OpState : (< o.m, (s_\sigma, f_\sigma) > \rightsquigarrow \ (s,f) \text{ implies } (\sigma' \mid s_{\sigma'} \rightarrow s, f_{\sigma'} \rightarrow f) \models P_2)))\end{array}$$

where $\sigma' = (\sigma \mid u_\sigma(D \rightarrow \delta))$

It is clear that when $D \in FV_D(P_1)$, the expression E^t occurs in $P_1[E^t/D]$. Using:

Lemma B.2 ($\sigma \models P$ and E^t occurs in P) implies $\exists\delta : |[E^t]|_\sigma = \delta$

which can be proven using structural induction on P it can be proven that $\exists\delta : |[E_t]|\sigma_1' = \delta$
Instantiating the universal quantifications above with $\sigma = \sigma_1$ and $\delta = |[E_t]|\sigma_1'$, conclude that

$\sigma_1'' \models P_1$ implies
$< o.m, (s\sigma_1, f\sigma_1) >$ is safe and
$\forall (s, f) \in OpState : (< o.m, (s\sigma_1, f\sigma_1) > \rightsquigarrow (s, f)$ implies $(\sigma_1'' \mid s\sigma_1'' \rightarrow s, f\sigma_1'' \rightarrow f) \models P_2)$

where $\sigma_1'' = (\sigma_1 \mid u\sigma_1(D \rightarrow |[E^t]|\sigma_1'))$
is a valid implication.

The next step of the proof shows that $\sigma_1' \models P_1[E_t/D]$ implies $\sigma_1'' \models P_1$. This is complicated by the fact that D might occur in E^t, and that E^t might occur in P_1. The solution is to introduce a fresh variable D' not occurring anywhere in the rule. We can guarantee D' being fresh by demanding that: $D' \notin FV_D(P_1)$ and $D' \notin FV_D(P_1[E^t/D])$. As D' does not occur free in $P_1[E^t/D]$ its value is of no importance to the truth value of $P_1[E^t/D]$. That means:

$\sigma_1' \models P_1[E^t/D]$ implies $\sigma_2 \models P_1[E^t/D]$, where $\sigma_2 = (\sigma_1' \mid u\sigma_1'(D' \rightarrow |[E^t]|\sigma_1'))$

The next implication relies on lemma B.1.
As σ_1' and σ_2 differ only on the mapping of D' it must be that $|[E^t]|\sigma_2 = |[E^t]|\sigma_1' = |[D']|\sigma_2$. Therefore:

$\sigma_2 \models P_1[E^t/D]$ implies $\sigma_2 \models (P_1[E^t/D])[D'/E^t]$

Using general rules of substitution one can conclude:

$\sigma_2 \models (P_1[E^t/D])[D'/E^t]$ implies $\sigma_2 \models P_1[D'/D][D'/E^t]$

Lemma B.1 works the other way around as well (following the same reasoning):

Lemma B.3 ($\sigma \models P[E_1/E_2]$ and $|[E_1]|_\sigma = |[E_2]|_\sigma$) implies $\sigma \models P$

Therefore, as $|[E^t]|\sigma_2 = |[D']|\sigma_2$ still holds:

$\sigma_2 \models P_1[D'/D][D'/E^t]$ implies $\sigma_2 \models P_1[D'/D]$

And because D does not occur in $P_1[D'/D]$:

$\sigma_2 \models P_1[D'/D]$ implies $\sigma_3 \models P_1[D'/D]$, where $\sigma_3 = (\sigma_2 \mid u\sigma_2(D \mapsto |[E^t]|\sigma_1'))$

This shows that $|[D]|\sigma_3 = |[E^t]|\sigma_1 = |[D']|\sigma_3$. Therefore, using lemma B.3:

$\sigma_3 \models P_1[D'/D]$ implies $\sigma_3 \models P_1$

Because D' does not occur in P_1, and because σ_2 and σ_1' differ only on the mapping of D', one can safely replace σ_2 with σ_1' in this implication

$\sigma_3 \models P_1$ implies $(\sigma_1' \mid u\sigma_1'(D \mapsto |[E^t]|\sigma_1')) \models P_1$

σ_1' and σ_1 differ only on the mapping of $\{D_1, \ldots, D_i\}$. But as $FV_D(P_1) = \{D\}$, it must be that if $D_j \in \{D_1, \ldots, D_i\}$ occurs in P_1 then $D_j = D$. As D is re-mapped the mapping of D_j by σ_1' has no effect on the truth value of P_1. Therefore:

$(\sigma_1' \mid u\sigma_1'(D \mapsto |[E^t]|\sigma_1')) \models P_1$ implies $(\sigma_1 \mid u\sigma_1(D \mapsto |[E^t]|\sigma_1')) \models P_1$

As $(\sigma_1 \mid u\sigma_1(D \mapsto |[E^t]|\sigma_1')) = \sigma_1''$ this concludes this step of the proof.
That means $\sigma_1' \models P_1[E_t/D]$ implies $\sigma_1'' \models P_1$ and $\sigma_1'' \models P_1$ implies $< o.m, (s\sigma_1, f\sigma_1) >$ is safe, satisfying half of the proof obligation. What remains to be proven is:

$\forall(s,f) \in OpState : (< o.m, (s\sigma_1, f\sigma_1) > \rightsquigarrow\ (s,f)$ implies $(\sigma_1' \mid s\sigma_1' \rightarrow s, f\sigma_1' \rightarrow f) \models P_2[E^t/D])$

Looking at the operational semantics defined in chapter 4 it can be seen that there is only one proof rule applicable to this situation. This rule establishes $< o.m, (s\sigma_1, f\sigma_1) > \rightsquigarrow\ (s\sigma_1, f)$ for some f. That means the proof obligation can be simplified to:

$\forall f : (< o.m, (s\sigma_1, f\sigma_1) > \rightsquigarrow\ (s\sigma_1, f)$ implies $(\sigma_1' \mid f\sigma_1' \rightarrow f) \models P_2[E^t/D])$

As was already proven, $\sigma_1' \models P_1[E^t/D]$ implies $\sigma_1'' \models P_1$ and
$\sigma_1'' \models P_1$ implies $\forall(s,f) \in OpState : (< o.m, (s\sigma_1, f\sigma_1) > \rightsquigarrow\ (s,f)$ implies $(\sigma_1'' \mid s\sigma_1'' \rightarrow s, f\sigma_1'' \rightarrow f) \models P_2)$

This can also be rewritten, to arrive at:

$\sigma_1' \models P_1[E^t/D]$ implies $\forall f : (< o.m, (s\sigma_1, f\sigma_1) > \rightsquigarrow\ (s\sigma_1, f)$ implies $(\sigma_1'' \mid f\sigma_1'' \rightarrow f) \models P_2)$

What remains to be proven is that $(\sigma_1'' \mid f\sigma_1'' \rightarrow f) \models P_2)$ implies $(\sigma_1' \mid f\sigma_1' \rightarrow f) \models P_2[E^t/D])$

Recall that the validity of $\{P_1\}\ o.m\ \{P_2\}$ implies that $FV_D(P_2) \subseteq FV_D(P_1)$.
That means that D' is also a fresh variable in P_2, so that

$(\sigma_1'' \mid f\sigma_1'' \rightarrow f) \models P_2$ implies $\sigma_4 \models P_2$, where $\sigma_4 = (\sigma_1'' \mid f\sigma_1'' \rightarrow f, u\sigma_1''(D' \rightarrow |[D]|\sigma_1''))$

As $|[D']|\sigma_4 = |[D]|\sigma_4$, lemma B.1 tells us that

$\sigma_4 \models P_2$ implies $\sigma_4 \models P_2[D'/D]$

$FV_D(P_2) \subseteq \{D\}$ implies $FV_D(P_2[D'/D]) \subseteq \{D'\}$. That means $D_j \in \{D_1, \ldots, D_i\}$ does not occur in $P_2[D'/D]$, which in turn ensures the validity of:

$\sigma_4 \models P_2[D'/D]$ implies $\sigma_5 \models P_2[D'/D]$, where $\sigma_5 = (\sigma_4 \mid u\sigma_4(\forall j.\ 1 \leq j \leq i.\ D_j \rightarrow \delta_j))$

$|[D']|\sigma_5 = |[D]|\sigma_1''$ by construction. However, $|[D]|\sigma_1'' = |[E^t]|\sigma_1'$, also by construction. The crucial point is now that a valuation $|[E^t]|_\sigma$ only relies on the *Store*, *UniVar* and *ExiVar* functions from σ. Note that σ_1' and σ_5 have identical *Store* and *ExiVar* functions (thanks to the fact that the statement $o.m$ only affects the *FieldHeap* function). $FV_D(P_1[E^t/D]) = \{D_1, \ldots, D_i\}$ implies $FV_D(E^t) \subseteq \{D_1, \ldots, D_i\}$. σ_1' and σ_5 map these variables to the same values, so that $|[E^t]|\sigma_1' = |[E^t]|\sigma_5$. That means that $|[D']|\sigma_5 = |[E^t]|\sigma_5$, which means that, using lemma B.1, the conclusion is:

$\sigma_5 \models P_2[D'/D]$ implies $\sigma_5 \models (P_2[D'/D])[E^t/D']$

Then, relying on general rules of substitution:

$\sigma_5 \models (P_2[D'/D])[E^t/D']$ implies $\sigma_5 \models P_2[E^t/D][E^t/D']$

D' has served its purpose and can be removed using lemma B.3:

$\sigma_5 \models P_2[E^t/D][E^t/D']$ implies $\sigma_5 \models P_2[E^t/D]$

σ_4 and $(\sigma_1'' \mid f\sigma_1'' \rightarrow f)$ differ only on D', which does not occur in $P_2[E^t/D]$. $(\sigma_1'' \mid f\sigma_1'' \rightarrow f)$ and $(\sigma_1' \mid f\sigma_1' \rightarrow f)$ differ only on D and on $\{D_1, \ldots, D_i\}$. As D does not occur in $P_2[E^t/D]$, and as σ_5 remaps $\{D_1, \ldots, D_i\}$ to their values in σ_1', conclude:

$\sigma_5 \models P_2[E^t/D]$ implies $(\sigma_1' \mid f\sigma_1' \rightarrow f) \models P_2[E^t/D]$

That concludes the proof of this rule.

B.5 Method Call

Proof of soundness for the inference rule for Method Call specifications is difficult, as it will have to be proven using induction on the depth of the recursion of the method. This section will prove the soundness of the version of the rule that can deal with recursion, leaving call-back as an in-principle straightforward extension. For reference, the inference rule (found in 10.6) is repeated:

$$\frac{\begin{array}{c}\text{Method Call with recursion:}\\ A \vdash \{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\} \\ (Pre_{c.m}[o/\mathbf{this}] * P) \Rightarrow (o \mapsto []^c * \mathbf{true})\end{array}}{\{Pre_{c.m}[o/\mathbf{this}] * P\}\ o.m\ \{Post_{c.m}[o/\mathbf{this}] * P\}}$$

We introduce the notation $[\{P_1\}\ S\ \{P_2\}]^{n,c.m}$, with $n \in \mathbb{N} \cup \infty$ and $\infty - 1 = \infty$. This is referred to as a *restricted specification.* Informally, this notation means that the specification $\{P_1\}\ S\ \{P_2\}$ holds for all executions with a maximum recursion depth n for method m of class c. This notion is formalized by the semantics of restricted specification:

with $FV_D(P_1) = \{D_1, \ldots, D_i\}$:

$[\{P_1\}\ S\ \{P_2\}]^{n,c.m}$ holds iff $\forall \sigma : (\forall \delta_1, \ldots, \delta_i : (\sigma' \models P_1$ implies
$\neg(< S, (s_\sigma, f_\sigma), (n, c.m) > \rightsquigarrow \mathbf{abort})$ and
$\forall (s, f) \in OpState : (< S, (s_\sigma, f_\sigma), (n, c.m) > \rightsquigarrow (s, f)$ implies $(\sigma' \mid s_{\sigma'} \rightarrow s, f_{\sigma'} \rightarrow f) \models P_2)))$

where $\sigma' = (\sigma \mid u_\sigma(\forall j.\ 1 \leq j \leq i.\ D_j \rightarrow \delta_j))$

The relation $\rightsquigarrow$ is very similar to $\rightsquigarrow$, but functions on an extended version of non-terminal configurations that includes $(n, c.m)$. Informally, $< S, (s, f), (n, c.m) > \rightsquigarrow (s', f')$ means that statement S leads from state (s, f) to terminal configuration (s', f') via an execution path with a maximum recursion depth n for method m as specified in class c. Such an n will be referred to as a *restrictor* for $c.m$. No rule is defined for executions with a higher recursion depth, ensuring that such configurations become stuck instead of leading to a terminal state. As could be expected, the definition of $\rightsquigarrow$ varies mainly from that of $\rightsquigarrow$ on the rules for restricted method call, which are:

RNAMC1 :

$$\frac{o \in dom(s) \quad s(o) \in dom(f) \quad m \in Methods(f(s(o))\rfloor_1) \quad f(s(o))\rfloor_1 = c \quad < Body(f(s(o))\rfloor_1, m), ((initS \mid \mathbf{this} \rightarrow s(o)), f), (n-1, c.m) > \rightsquigarrow (s', f')}{< o.m, (s, f), (n, c.m) > \rightsquigarrow (s, f')}$$

RNAMC2 :

$$\frac{o \in dom(s) \quad s(o) \in dom(f) \quad m \in Methods(f(s(o))\rfloor_1) \quad f(s(o))\rfloor_1 \neq c \quad < Body(f(s(o))\rfloor_1, m), ((initS \mid \mathbf{this} \rightarrow s(o)), f), (n, c.m) > \rightsquigarrow (s', f')}{< o.m, (s, f), (n, c.m) > \rightsquigarrow (s, f')}$$

RAMC :

$$\frac{\begin{array}{c}o \notin dom(s)\ or\ s(o) \notin dom(f)\ or\ m \notin Methods(f(s(o))\rfloor_1)\ or \\ (f(s(o))\rfloor_1 \neq c\ and\ < Body(f(s(o))\rfloor_1, m), ((initS \mid \mathbf{this} \rightarrow s(o)), f), (n, c.m) > \rightsquigarrow \mathbf{abort})\ or \\ (f(s(o))\rfloor_1 = c\ and\ < Body(f(s(o))\rfloor_1, m), ((initS \mid \mathbf{this} \rightarrow s(o)), f), (n-1, c.m) > \rightsquigarrow \mathbf{abort})\end{array}}{< o.m, (s, f), (n, c.m) > \rightsquigarrow \mathbf{abort}}$$

All other rules are virtually unchanged, as they do not use the extended configuration's additional information. The only change is that every configuration is replaced by an extended configuration

with $(n, c.m)$ as the additional information. As an example, the rules for restricted sequential composition are:

$$\frac{< S_1, (s_1, f_1), (n, c.m) > \rightsquigarrow\ (s_2, f_2) \quad < S_2, (s_2, f_2), (n, c.m) > \rightsquigarrow\ (s_3, f_3)}{< S_1; S_2, (s_1, f_1), (n, c.m)) > \rightsquigarrow\ (s_3, f_3)}$$

$$\frac{< S_1, (s, f), (n, c.m) > \rightsquigarrow\ \textbf{abort}}{< S_1; S_2, (s, f), (n, c.m) > \rightsquigarrow\ \textbf{abort}}$$

$$\frac{< S_1, (s_1, f_1), (n, c.m) > \rightsquigarrow\ (s_2, f_2) \quad < S_2, (s_2, f_2), (n, c.m) > \rightsquigarrow\ \textbf{abort}}{< S_1; S_2, (s, f), (n, c.m) > \rightsquigarrow\ \textbf{abort}}$$

We claim (without a formal proof) that the relations $\rightsquigarrow$ and $\leadsto$ are equivalent for $n = \infty$, as $\infty - 1 = \infty$. This supports the following lemma:

Lemma B.4 $[\{P_1\}\ S\ \{P_2\}]^{\infty, c.m}$ iff $\{P_1\}\ S\ \{P_2\}$

That means that when it is proven that, for all n:

$$\frac{\begin{array}{c} A \vdash \{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\} \\ (Pre_{c.m}[o/\textbf{this}] * P) \Rightarrow (o \mapsto []^c * \textbf{true}) \end{array}}{[\{Pre_{c.m}[o/\textbf{this}] * P\}\ o.m\ \{Post_{c.m}[o/\textbf{this}] * P\}]^{n, c.m}}$$

is sound, the original inference rule is proven sound as well, as when
$[\{Pre_{c.m}[o/\textbf{this}] * P\}\ o.m\ \{Post_{c.m}[o/\textbf{this}] * P\}]^{\infty, c.m}$ is valid,
$\{Pre_{c.m}[o/\textbf{this}] * P\}\ o.m\ \{Post_{c.m}[o/\textbf{this}] * P\}$ is valid as well.

The rule above is proven using induction on n, with the rule as induction hypothesis.
Proof starts with a general section that is used in both the base and the step case.

general section
First assume that the premise of the rule holds.
Then, assume without loss of generality that $FV_D(Pre_{c.m}[o/\textbf{this}] * P) = \{D_1, \ldots, D_i\}$.
Furthermore, assume a τ and $\delta_1, \ldots, \delta_i$ such that:

$\tau' \models Pre_{c.m}[o/\textbf{this}] * P$, where $\tau' = (\tau \mid u_\tau(\forall j.\ 1 \leq j \leq i.\ D_j \rightarrow \delta_j))$

base case
The base case is $n = 0$. The proof obligation for this case under the general assumptions above is given by the semantics of restricted specifications:

$\neg(< o.m, (s_\tau, f_\tau), (0, c.m) >)\ \rightsquigarrow\ \textbf{abort}$ and
$\forall (s, f) \in OpState:\ (< o.m, (s_\tau, f_\tau), (0, c.m) >)\ \rightsquigarrow\ (s, f)$ implies
$(\tau' \mid s_{\tau'} \rightarrow s, f_{\tau'} \rightarrow f) \models Post_{c.m}[o/\textbf{this}] * P)$

There are three operational rules that could apply to $< o.m, (s_\tau, f_\tau), (0, c.m) >$. One of these rules, **RAMC**, leads to **abort** and could endanger the safety of this configuration. The premise of this rule (for the general case) is:

$o \notin dom(s)$ *or* $s(o) \notin dom(f)$ *or* $m \notin Methods(f(s(o))\rfloor_1)$ *or*
$(f(s(o))\rfloor_1 \neq c$ *and* $< Body(f(s(o))\rfloor_1, m), ((initS \mid \textbf{this} \rightarrow s(o)), f), (n, c.m) > \rightsquigarrow\ \textbf{abort})$ *or*
$(f(s(o))\rfloor_1 = c$ *and* $< Body(f(s(o))\rfloor_1, m), ((initS \mid \textbf{this} \rightarrow s(o)), f), (n - 1, c.m) > \rightsquigarrow\ \textbf{abort})$

However, $\tau' \models Pre_{c.m}[o/\textbf{this}] * P$. As it was assumed that the premise of the inference rule holds, $(Pre_{c.m}[o/\textbf{this}] * P) \Rightarrow (o \mapsto []^c * \textbf{true})$. As a result, $\tau' \models o \mapsto []^c * \textbf{true}$. The semantics of $\mapsto$ then show that $o \in dom(s_{\tau'})$ and $s_{\tau'}(o) \in dom(f_{\tau'})$. τ' only differs from τ on the *Univar* function so that $s_{\tau'} = s_\tau$ and $f_{\tau'} = f_\tau$. It also shows that $f_\tau(s_\tau(o))\rfloor_1 = c$. As the specification $\{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\}$ is required to be unprovable when there is no method m in class c (see

10.6) and $A \vdash \{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\}$ was assumed, $m \in Methods(f(s(o))\rfloor_1)$ must be true.

That leaves $< Body(f_\tau(s_{\tau'}(o))\rfloor_1, m), ((initS \mid \textbf{this} \rightarrow s(o)), f_{\tau'}), (0-1, c.m) > \rightsquigarrow$ **abort**. However, as $-1 \notin \mathbb{N}$, this is not a well-formed extended configuration and therefore does not lead to a terminal state. This proves safety for $n = 0$. Furthermore, **RNAMC1** requires this same configuration to lead to a terminal state to be used. Finally, **RNAMC2** cannot be applied because of the clause $f_\tau(s_\tau(o))\rfloor_1 \neq c$, of which the opposite was shown to hold above. That shows that this configuration becomes stuck (as could be expected from an execution of a method call with recursion depth 0), which satisfies the second requirement of the proof obligation and concludes the case $n = 0$.

step case $n+1$
In the induction step $n+1$ the induction hypothesis, which is referred to as $IH(n)$, is assumed to hold.
Under the assumptions from the general section the proof obligation for the step case is:

$\neg(< o.m, (s_\tau, f_\tau), (n+1, c.m) >) \rightsquigarrow$ **abort** and
$\forall (s, f) \in OpState:\ (< o.m, (s_\tau, f_\tau), (n+1, c.m) >) \rightsquigarrow (s, f)$ implies
$(\tau' \mid s_{\tau'} \rightarrow s, f_{\tau'} \rightarrow f) \models Post_{c.m}[o/\textbf{this}] * P)$

Analog to the base case, the configuration $< o.m, (s_\tau, f_\tau), (n+1, c.m) >$ has three operational rules that could apply to it. The exact same reasoning as used in that case is used here to show that $o \in dom(s_\tau)$, $s_\tau(o) \in dom(f_\tau)$, $m \in Methods(f_\tau(s_\tau(o))\rfloor_1)$ and $f_\tau(s_\tau(o))\rfloor_1 = c$. That excludes the application of **RNAMC2**, and leaves two possibilities. The transition from the state $< Body(c, m), ((initS \mid \textbf{this} \rightarrow s_\tau(o)), f_\tau), (n, c.m) >$ is crucial to the application of either one.

This transition must be proven not to lead to **abort** to prevent the possible application of the aborting rule for method call **RAMC**. Additionally, when it does lead to a terminal configuration, application of the rule for non-aborting method call **RNAMC1** must produce the desired result:

$\neg(< Body(c, m), ((initS \mid \textbf{this} \rightarrow s_\tau(o)), f_\tau), (n, c.m) > \rightsquigarrow$ **abort**) and
$\forall (s, f) \in OpState:\ (< Body(c, m), ((initS \mid \textbf{this} \rightarrow s_\tau(o)), f_\tau), (n, c.m) >) \rightsquigarrow (s, f)$ implies
$(\tau' \mid f_{\tau'} \rightarrow f) \models Post_{c.m}[o/\textbf{this}] * P)$

To prove that these two requirements hold, attention is turned to three of the assumptions made so far:

$A \vdash \{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\}$
$(Pre_{c.m}[o/\textbf{this}] * P) \Rightarrow (o \mapsto []^c * \textbf{true})$
$IH(n)$

These assumptions might seem to imply the validity of $\{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\}$. However, this is only the case when A is proven to hold, which is an even stronger proof obligation then the one started with.

Instead, they are used to show the validity of $[\{Pre_{c.m} * P\}\ Body_{c.m}\ \{Post_{c.m}\}]^{n,c.m}$. Once that is established, it is proven that this restricted specification implies the requirements stated above.

sub-proof for $[\{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\}]^{n,c.m}$
Proof starts with the a lemma that states that when a specification is proven valid up to a certain recursion depth of a method, it is valid up to all lesser recursion depths as well:

Lemma B.5 $[\{P_1\}\ S\ \{P_2\}]^{n,c.m}$ implies $\forall j : 0 \leq j \leq n : [\{P_1\}\ S\ \{P_2\}]^{j,c.m}$

Induction on the shape of shape of the derivation tree can prove this lemma.

Lemma B.4 says that when $\{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\}$ holds, $[\{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\}]^{\infty,c.m}$ holds as well. Applying the lemma above then proves:

$A \vdash \{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\}$ implies $A \vdash [\{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\}]^{n,c.n}$

Assumption A cannot be proven to hold. However, we conjecture that whenever $A \vdash [\{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\}]^{n,c.n}$, a weaker assumption A' is also sufficient, and then show that this A' holds to arrive at our conclusion. The weaker A' is (the singleton set of):

$$\frac{(Pre_{c.m}[o'/\mathbf{this}] * P') \Rightarrow (o' \mapsto []^c * \mathbf{true})}{[\{Pre_{c.m}[o'/\mathbf{this}] * P'\}\ o.m\ \{Post_{c.m}[o'/\mathbf{this}] * P'\}]^{n,c.m}}$$

So, the claim is that whenever $[\{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\}]^{n,c.n}$ can be proven under assumption A, it can be proven under an assumption that restricts A to n as well.

To see why this is true look at the proof obligation for $[\{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\}]^{n,c.n}$:

with $FV_D(Pre_{c.m}) = \{D_1, \ldots, D_i\}$:

$[\{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\}]^{n,c.m}$ holds iff $\forall \sigma : (\forall \delta_1, \ldots, \delta_i : (\sigma' \models Pre_{c.m}$ implies
$\neg(< Body_{c.m}, (s_\sigma, f_\sigma), (n, c.m) > \rightsquigarrow$ **abort**) and
$\forall (s, f) \in OpState :\ (< Body_{c.m}, (s_\sigma, f_\sigma), (n, c.m) > \rightsquigarrow\ (s, f)$ implies
$(\sigma' \mid s_{\sigma'} \rightarrow s, f_{\sigma'} \rightarrow f) \models Post_{c.m})))$

where $\sigma' = (\sigma \mid u_\sigma(\forall j.\ 1 \leq j \leq i.\ D_j \rightarrow \delta_j))$

Such a proof obligation relies on derivations of $< Body_{c.m}, (s_\sigma, f_\sigma), (n, c.m) > \rightsquigarrow\ t$ for terminal configurations t. Assumption A allows such derivations to use properties of $< o.m, (s_\sigma, f_\sigma), (\infty, c.m) > \rightsquigarrow\ t$ as axioms for certain σ. The crucial point is that there are no rules besides lemma B.5 where the restrictor of $c.m$ of a configuration in the premise is larger than that restrictor in the conclusion. That means that in no branch of a proof tree for $< Body_{c.m}, (s_\sigma, f_\sigma), (n, c.m) > \rightsquigarrow\ t$ a restrictor larger than n is required, and thus that A' is sufficient as an assumption whenever A is.

To show that A' does indeed hold, $IH(n)$ is repeated:

$$\frac{A \vdash \{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\} \qquad (Pre_{c.m}[o/\mathbf{this}] * P) \Rightarrow (o \mapsto []^c * \mathbf{true})}{[\{Pre_{c.m}[o/\mathbf{this}] * P\}\ o.m\ \{Post_{c.m}[o/\mathbf{this}] * P\}]^{n,c.m}}$$

As can be seen above, both premises of this rule have been assumed. That means

$$[\{Pre_{c.m}[o/\mathbf{this}] * P\}\ o.m\ \{Post_{c.m}[o/\mathbf{this}] * P\}]^{n,c.m}$$

can be concluded. Using the rule of variable substitution as defined in 10.2 the following holds as well:

$$[(\{Pre_{c.m}[o/\mathbf{this}] * P\}\ o.m\ \{Post_{c.m}[o/\mathbf{this}] * P\})[o'/o]]^{n,c.m}$$

As $Pre_{c.m}$ and $Post_{c.m}$ are known not to have any free temporary variables, this implies:

$$[\{Pre_{c.m}[o'/\mathbf{this}] * P[o'/o]\}\ o'.m\ \{Post_{c.m}[o'/\mathbf{this}] * P[o'/o]\}]^{n,c.m}$$

Also assumed is $(Pre_{c.m}[o/\mathbf{this}] * P) \Rightarrow (o \mapsto []^c * \mathbf{true})$

Now rely on the axiom

Axiom B.6 $(P_1 \Rightarrow P_2) \Leftrightarrow (P_1[E_1/E_2] \Rightarrow P_2[E_1/E_2])$

to conclude that $(Pre_{c.m}[o/\mathbf{this}] * P)[o'/o] \Rightarrow (o \mapsto []^c * \mathbf{true})[o'/o]$, from which is inferred: $(Pre_{c.m}[o'/\mathbf{this}] * P[o'/o]) \Rightarrow (o' \mapsto []^c * \mathbf{true})$

Another axiom is introduced:

Axiom B.7 $\dfrac{A \quad B}{\left(\dfrac{A}{B}\right)}$

Already shown to hold were $(Pre_{c.m}[o'/\mathbf{this}] * P[o'/o]) \Rightarrow (o' \mapsto []^c * \mathbf{true})$ and $[\{Pre_{c.m}[o'/\mathbf{this}] * P[o'/o]\}\ o'.m\ \{Post_{c.m}[o'/\mathbf{this}] * P[o'/o]\}]^{n,c.m}$. When $P[o'/o]$ is replaced by P', without loss of generality as there are no restrictions on P's shape, the axiom can be applied to conclude the proof that A' holds.

The proof continues with the introduction of yet another axiom:

Axiom B.8 $\dfrac{A \vdash B \quad A}{B}$

It was shown that A' holds and that $A' \vdash [\{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\}]^{n,c.m}$. That means the axiom can be applied to arrive at the conclusion of this sub-proof.
end of sub-proof for $[\{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\}]^{n,c.m}$

Now assume without loss of generality that $FV_D(Pre_{c.m}) = \{D_1, \ldots, D_i\}$
The semantics of restricted specifications show that:

$$
\begin{array}{l}
[\{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\}]^{n,c.m} \text{ holds iff } \forall \sigma : (\forall \delta_1, \ldots, \delta_i : (\sigma' \models Pre_{c.m} \text{ implies} \\
\quad \neg(< Body_{c.m}, (s_\sigma, f_\sigma), (n, c.m) > \rightsquigarrow \mathbf{abort}) \text{ and} \\
\quad \forall (s, f) \in OpState : \ (< Body_{c.m}, (s_\sigma, f_\sigma), (n, c.m) > \rightsquigarrow \ (s, f) \text{ implies} \\
\qquad\qquad\qquad\qquad (\sigma' \mid s_{\sigma'} \rightarrow s, f_{\sigma'} \rightarrow f) \models Post_{c.m})))
\end{array}
$$

where $\sigma' = (\sigma \mid u_\sigma(\forall j.\ 1 \leq j \leq i.\ D_j \rightarrow \delta_j))$

Recall that $\tau' \models Pre_{c.m}[o/\mathbf{this}] * P$. The semantics of $*$ then say that there are f_1 and f_2, where f_1 and f_2 have disjoint domains and $f_{\tau'} = f_1 \cdot f_2$, such that:

$$(\tau' \mid f_{\tau'} \rightarrow f_1) \models Pre_{c.m}[o/\mathbf{this}] \text{ and } (\tau' \mid f_{\tau'} \rightarrow f_2) \models P$$

As $Pre_{c.m}$ is known not to contain any temporary variables and **this** is replaced by o in $Pre_{c.m}[o/\mathbf{this}]$, this proposition's truth value is independent of the *Store* function with the exception of the mapping of o. Therefore:

$$(\tau' \mid f_{\tau'} \rightarrow f_1) \models Pre_{c.m}[o/\mathbf{this}] \text{ implies } (\tau' \mid f_{\tau'} \rightarrow f_1, s_{\tau'} \rightarrow initS(o \rightarrow s_{\tau'}(o))) \models Pre_{c.m}[o/\mathbf{this}]$$

And, as **this** does not occur in $Pre_{c.m}[o/\mathbf{this}]$, also:

$$
\begin{array}{l}
(\tau' \mid f_{\tau'} \rightarrow f_1, s_{\tau'} \rightarrow initS(o \rightarrow s_{\tau'}(o))) \models Pre_{c.m}[o/\mathbf{this}] \text{ implies} \\
(\tau' \mid f_{\tau'} \rightarrow f_1, s_{\tau'} \rightarrow initS(o \rightarrow s_{\tau'}(o), \mathbf{this} \rightarrow s_{\tau'}(o))) \models Pre_{c.m}[o/\mathbf{this}]
\end{array}
$$

In the state of the righthand side of the implication above, o and **this** are mapped to the same value. That means lemma B.3 can be used to prove:

$$
\begin{array}{l}
(\tau' \mid f_{\tau'} \rightarrow f_1, s_{\tau'} \rightarrow initS(o \rightarrow s_{\tau'}(o), \mathbf{this} \rightarrow s_{\tau'}(o))) \models Pre_{c.m}[o/\mathbf{this}] \text{ implies} \\
(\tau' \mid f_{\tau'} \rightarrow f_1, s_{\tau'} \rightarrow initS(o \rightarrow s_{\tau'}(o), \mathbf{this} \rightarrow s_{\tau'}(o))) \models Pre_{c.m}
\end{array}
$$

Furthermore, as o does not occur in $Pre_{c.m}$:

$$
\begin{array}{l}
(\tau' \mid f_{\tau'} \rightarrow f_1, s_{\tau'} \rightarrow initS(o \rightarrow s_{\tau'}(o), \mathbf{this} \rightarrow s_{\tau'}(o))) \models Pre_{c.m} \text{ implies} \\
(\tau' \mid f_{\tau'} \rightarrow f_1, s_{\tau'} \rightarrow initS(\mathbf{this} \rightarrow s_{\tau'}(o))) \models Pre_{c.m}
\end{array}
$$

This important conclusion allows one to instantiate the universal quantification over σ from the semantics of the restricted specification $[\{Pre_{c.m}\}\ Body_{c.m}\ \{Post_{c.m}\}]^{n,c.m}$ given above to conclude that:

$\neg(< Body_{c.m}, (initS(\mathbf{this} \rightarrow s_{\tau'}(o)), f_1), (n, c.m) > \rightsquigarrow \mathbf{abort})$ and
$\forall (s, f) \in OpState:\ (< Body_{c.m}, (initS(\mathbf{this} \rightarrow s_{\tau'}(o)), f_1), (n, c.m) > \rightsquigarrow (s, f)$ implies
$(\tau' \mid s_{\tau'} \rightarrow s, f_{\tau'} \rightarrow f) \models Post_{c.m}))$

The proof relies on two important properties of the language as stated at the end of section 2 of [8] (slightly modified to apply to our language and the extended specifications). The first says that when an extended configuration with a *FieldHeap* function f leads to **abort**, any similar configuration with a smaller *FieldHeap* function will lead to **abort** as well:

for all $f_1 \subseteq f$

if $< S, (s, f), (n, c.m) > \rightsquigarrow \mathbf{abort}$ then $< S, (s, f_1), (n, c.m) > \rightsquigarrow \mathbf{abort}$

This property is used to prove the first of the two remaining proof obligations for the step case, which is:

$\neg(< Body(c, m), ((initS \mid \mathbf{this} \rightarrow s_\tau(o)), f_\tau), (n, c.m) > \rightsquigarrow \mathbf{abort})$

Now assume that $< Body(c, m), ((initS \mid \mathbf{this} \rightarrow s_\tau(o)), f_\tau), (n, c.m) > \rightsquigarrow \mathbf{abort}$. Then, as $f_\tau = f_1 \cdot f_2$ and thus $f_1 \subseteq f_\tau$, also $< Body(c, m), ((initS \mid \mathbf{this} \rightarrow s_\tau(o)), f_1), (n, c.m) > \rightsquigarrow \mathbf{abort}$. Note however that $Body(c, m)$ is a different notation for $Body_{c.m}$ and that $s_\tau = s_{\tau'}$. That means that this contradicts the conclusion reached above that this configuration does not lead to **abort** (note that $s_\tau = s_{\tau'}$), and thereby proves the obligation.

The only remaining proof obligation is:

$\forall (s, f) \in OpState:\ (< Body(c, m), ((initS \mid \mathbf{this} \rightarrow s_\tau(o)), f_\tau), (n, c.m) >) \rightsquigarrow (s, f)$ implies
$(\tau' \mid f_{\tau'} \rightarrow f) \models Post_{c.m}[o/\mathbf{this}] * P)$

This can be proven to hold using the property:

for $f_1 \# f_2$ and $f = f_1 \cdot f_2$

if $< S, (s, f), (n, c.m) > \rightsquigarrow (s', f')$ then
$< S, (s, f_1), (n, c.m) > \rightsquigarrow \mathbf{abort}$ or
$< S, (s, f_1), (n, c.m) > \rightsquigarrow (s', f_1')$ where $f_1' \# f_2$ and $f' = f_1' \cdot f_2$

Now assume $< Body(c, m), ((initS \mid \mathbf{this} \rightarrow s_\tau(o)), f_\tau), (n, c.m) >) \rightsquigarrow (s, f)$. As $f_1 \# f_2$ and $f_\tau = f_1 \cdot f_2$ the property can be applied:

$< Body(c, m), ((initS \mid \mathbf{this} \rightarrow s_\tau(o)), f_1), (n, c.m) > \rightsquigarrow \mathbf{abort}$ or
$< Body(c, m), ((initS \mid \mathbf{this} \rightarrow s_\tau(o)), f_1), (n, c.m) > \rightsquigarrow (s', f_1')$ where $f_1' \# f_2$ and $f = f_1' \cdot f_2$

Once again turn to the conclusion reached above to see that
$\neg(< Body(c, m), ((initS \mid \mathbf{this} \rightarrow s_\tau(o)), f_1), (n, c.m) > \rightsquigarrow \mathbf{abort})$, meaning the second clause above must hold. Instantiating the second part of the conclusion above with that clause gives:

$(\tau' \mid s_{\tau'} \rightarrow s', f_{\tau'} \rightarrow f_1') \models Post_{c.m}$

As $Post_{c.m}$ does not contain any temporary variables, and the value of **this** is always the same in both the start and end configuration (as none of the operational rules change its value), this implies:

$(\tau' \mid s_{\tau'} \rightarrow initS(\mathbf{this} \rightarrow s_\tau(o)), f_{\tau'} \rightarrow f_1') \models Post_{c.m}$

As o is not in $Post_{c.m}$, its value can be mapped to that of **this** in the state above, after which lemma B.1 can be used:

$(\tau' \mid s_{\tau'} \rightarrow initS(o \rightarrow s_\tau(o), \mathbf{this} \rightarrow s_\tau(o)), f_{\tau'} \rightarrow f_1') \models Post_{c.m}[o/\mathbf{this}]$

The only *Store* value $Post_{c.m}[o/\mathbf{this}]$ depends on is that of o. That means (as $s_\tau = s_{\tau'}$):

$(\tau' \mid s_{\tau'} \rightarrow initS(o \rightarrow s_\tau(o), \mathbf{this} \rightarrow s_\tau(o)), f_{\tau'} \rightarrow f_1') \models Post_{c.m}[o/\mathbf{this}]$ implies
$(\tau' \mid f_{\tau'} \rightarrow f_1') \models Post_{c.m}[o/\mathbf{this}]$

The final step of this proof uses the semantics of $*$, $f = f_1' \cdot f_2$, and the assumption that $(\tau' \mid f_{\tau'} \rightarrow f_2) \models P$ to conclude:

$(\tau' \mid f_{\tau'} \rightarrow f) \models Post_{c.m}[o/this] * P$

This concludes the proof of the step case of the induction proof and therefore the proof of the induction. Although realizing this is not a full formal proof, we consider this a good enough proof outline to be convinced of the soundness of the inference rule.