Universidade Federal de Pernambuco
Centro de Informática

Pós-graduação em Ciência da Computação

Tese de Doutorado

A Unifying Theory of Object-Orientation

por

Thiago Luiz Vieira de Lima Santos

Orientador: Prof. Dr. Augusto Sampaio
Co-Orientadora: Profa. Dra. Ana Cavalcanti

Resumo

O uso de linguagens orientadas a objetos em diversos ramos de aplicações é comum nos dias atuais. O interesse em tais linguagens e nos diferentes domínios nos quais elas vêm sendo empregadas têm alavancado a pesquisa e o uso de métodos formais para descrever precisamente o comportamento de programas neste paradigma. Trabalhos relacionados a outros paradigmas, como o imperativo e o funcional, têm sido progressivamente revisados, ou estendidos, para tratar características presentes nas linguagens orientadas a objetos, como, por exemplo, o encapulamento, a ligação dinâmica, o polimorfismo, e, em alguns casos, herança comportamental. De um modo geral, já existe um razoável conjunto de formalismos usados para descrever as linguages, orientadas a objetos ou não, mas suas diferentes sintaxes, ou principalmente seus diferentes *frameworks* semânticos, dificultam uma combinação ou comparação direta entre as linguages descritas.

A Unifying Theories of Programming (UTP) é um *framework* semântico proposto por Hoare e He para permitir que diferentes paradigmas sejam descritos usando uma mesma base, no caso, predicados alfabetizados. Cada conceito do paradigma que está sendo estudado é descrito como um predicado lógico, e ao conjunto de construtores e predicados relacionados a um paradigma é dado o nome de teoria. A definição de diferentes teorias usando um formalismo comum, que já foi mostrado como sendo poderoso o suficiente para modelar desde uma linguagem seqüencial simples até interações entre processos concorrentes que se comunicam, permite que elas sejam diretamente combinadas ou comparadas.

Neste trabalho é criada uma teoria para orientação a objetos com sua semântica definida usando-se os predicados alfabetizados da UTP. Usando esta teoria, que é uma combinação das teorias de designs e procedimentos de alta ordem apresentados por Hoare e He, descrevemos conceitos como herança e ligação dinâmica. Posteriormente, combinamos esta teoria com uma que trata de ponteiros para, finalmente, propormos e provarmos leis para programas orientados a objetos. É importante ressaltar entretanto que, como estamos trabalhando com semântica e não com uma linguagem específica, as leis se aplicam para quaisquer linguagens orientadas a objetos que possam ser descritas usando a semântica proposta.

Palavras-chave: Orientação a objetos; UTP; Semântica Relacional; Leis Algébricas; Integração

de Linguages.

Abstract

Currently, the use of object-oriented languages in a wide range of applications is very common. The interest in such languages, and the different domains in which they have been applied, have encouraged the research and the adoption of formalisms to describe precisely the behaviour of programs in this paradigm. Results related to other paradigms, such as imperative and functional, have been progressively revised, or extended, to cope with object-oriented features like encapsulation, dynamic binding, polymorphism, and, in some cases, behavioural inheritance. There is a relatively large set of such formalisms used to describe different languages, object-oriented or not, but their different syntaxes, and semantic frameworks difficult their straight combination or comparison.

The Unifying Theories of Programming (UTP) is a semantic framework proposed by Hoare and He to allow different paradigms to be described under the same basis of alphabetised predicates. Each paradigm concept under consideration is described by logical predicates, and the set of constructs and predicates related to a given paradigm is named a theory. Defining different theories using a common formalism, already known to be powerful enough to model from a simple sequential language to interactive and concurrent communicating processes, it is possible to combine and compare them.

This work defines a theory of object-orientation with its semantics defined in terms of the UTP alphabetised predicates. The description of concepts, such as inheritance and dynamic binding, is provided; using this theory, which is a combination of the theory of designs and higher-order procedures presented by Hoare and He. Finally, we combine this theory with another that address pointers and then we propose and prove laws for object-oriented programs. Since we are working with semantics, not with the syntaxes, these laws apply to any object-oriented language that share the concepts present in our theory.

Keywords: Object-orientation; UTP; Relational Semantics; Algebraic Laws; Integration of Languages.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we present an introduction to our research motivation and topics, and describe the overall structure and objectives of this thesis. In Section 1.1 we provide an overview of the context. Next, in Section 1.2 we highlight our objectives, and in Section 1.3 we describe the thesis general structure.

1.1 Overview

Since object-oriented languages have been widely used in different domains and applications, there has been a lot of interest in understanding and describing the meaning of object-oriented programs. Approaches like operational [Plo81], denotational [Sch86], and algebraic semantics [HHJ+87] have been used to describe different kinds of languages, and how their concepts are related.

The wide-spread use of the Unifying Modeling Language [BJR98, OMG97] is one of the factors that contributed to the increasing number of descriptions of systems using concepts common to the object-oriented paradigm, and further development of these systems using an object-oriented language like C++ [Str85] or Java [GJSB00]. In this context, more specifically in language design, imperfections like the ones found in Eiffel's [Mey92, Coo89] type system has revealed a strong need to introduce some sort of formalism in such languages, or to analyse them, to avoid inconsistencies with static or runtime types, or to allow their reliable use in critical application domains.

Not only for object-oriented languages, but in the general case, the industry has become aware that the use of formalism at some stages of the software development process can avoid program errors or, at least, improve software reliability. The inclusion of formal method techniques into software development processes has been an active research topic [Jon90, Abr96, BRL03, BCD+06]. It is important to remark, however, that there are other tools, usually less formal, that can minimize errors in the final software product like, for example, peer reviews and testing strategies already included in software development processes such as Extreme Programming (XP) [DW06].

Basically, there are two well-known approaches to introduce formalism in a software development process. In a *constructive approach*, the system construction starts from an abstract formal specification and by the application of refinement steps, based on a refinement calculus, a correct system is developed, with concrete and implementable abstractions [Mor94, BvW98], such as in the B method [Abr96] and in VDM [Jon90]. Each refinement step aims to reduce abstraction to allow machine implementation.

In a *verification approach* [GH93], where some properties about a given system are specified independently of its implementation, and after, or even during the implementation phase, these properties are formally verified in order to validate such an implementation. In some cases, these specifications are just a starting point to the software development. They are expected to allow developers to have a better understanding of what they are going to implement and, perhaps, to foresee the general behaviour of the system. Actually, the introduction of specification during software implementation has become a reality with the use of specification languages that are closer to the implementation languages; for example, for Java we have the Java Modeling Language (JML) [LBR99, BCC+03], which allows developers to introduce specification into Java programs. Another example of integration between host and specification languages is the Boogie methodology [BCD+06], where programs in C# [Mic07a] are extended with specifications, resulting in a language named Spec# [BLS05].

As usual, the approach to be followed depends on: cost, time budgets, application domain, and, moreover, on the set of available tools. In the following, we cite some examples of efforts to introduce formalism in practical software development processes:

- there are many attempts to formalise visual notations like UML [FEL97, Eva98, BF98, RG98, EFLR99, LB98, LBE00, FOW01, PMP01], or at least to enrich it with precise descriptions by means of annotations [KW98, RG02] – allowing code generation in better conformance with the graphical representation. Another example is the definition of a visual language with a formal background like Alloy [Dan02, Jac06] and its use for reasoning about object-oriented programs;
- for Java, a lot of work has been done to prove its type-safety [DE98, vON99, Sym99, PHM99], or to describe formally the Java Virtual Machine behaviour [Qia99, vO00, Hui01] – to avoid static and dynamic problems with its type system or to predict system behaviour;
- as already mentioned, there are approaches concerned with the introduction of specifications into source code [LBR99, BCC+03, BCD+06], during the implementation phase.

The actual fact is that the formalism can be introduced using different frameworks and tools to support different needs, at different points of the software development process, but the general idea is that with formalism one can reason about programs. This challenge is pertinent for different programming paradigms and languages.

The context of this thesis is the semantics of object-oriented languages, where we define a theory in the Unifying Theories of Programming (UTP) [HH98] that allows different languages to be mapped into it, and thus, properties (i.e. laws) can be proved. The introduced semantics can be used in both approaches, *constructive* or *verification*. At some stage, these approaches need a semantic framework (in our case we adopt the UTP) and a theory for object-orientation (our theory can model object-oriented features) to allow reasoning about object-oriented programs. Moreover, the theory integration is essential to the UTP and thus we can combine the developed theory with others. In Chapter 4 we present the theory for object-orientation which is already a combination of designs and higher-order procedures, and later, in Chapter 5, we provide an integration of this theory with a theory of pointers [CHW06] (extended in [HCW07]).

1.1.1 Related Work

The formalisation of object-oriented systems has been under great evolution, but there are many problems and features still to be handled [LLM06b, LLM06a]. In the sequel we introduce some works on formalisation that are described in more detail in Chapter 2.

In [AL97] Abadi and Leino introduce a logic similar to Hoare logic [Hoa69] for reasoning about object-oriented programs and prove its soundness. Their approach, however, lacks some of the concepts common in object-oriented languages such as dynamic binding. In [AL03] the authors revised [AL97] to include proofs and more recent related works. In [Lei98] Leino extends the work presented in [AL97] to cope with recursive types and dynamic binding. He also provides a systematic fine-grained separation of object-oriented constructs, which we adopt here.

The work presented in [MPH97], where Müller and Poetzsch-Heffter provide a description of an object-oriented language named KOOL (**k**ernel of **o**bject-**o**riented **l**anguages), introduces some general techniques for object-oriented program verification that are further detailed in [PHM98]. The set of features handled, however, is closer to real programming languages than those presented in [AL97]; it includes dynamic binding and inheritance. Nevertheless, unlike [AL97], soundness is not addressed. In [PHM99] a high-order logic is used to prove the soundness of a structural operational semantic defined for the language presented in [PHM98], extended with encapsulation.

In [Ral00, BMvW00], an extension of [MS97], Back *et al.* concentrate on the notion of refinement to allow object substitutions; basically a refinement calculus [BvW98] is extended to handle object-oriented features. The subtyping relation is defined and used to show that, if a subclass is a refinement of its superclass, a subtype instance can correctly replace an instance of the original superclass, the subtype polymorphism. Behavioural aspects of subtyping presented in [Ame90, LW94, Mey97] are also considered.

Cavalcanti and Naumann [CN00] presents a language called Refinement Object-Oriented Language (ROOL) which is a subset of sequential Java combined with some refinement constructs such

as Morgan's specification statements [Mor94]. In their work, a semantics for ROOL is provided in terms of weakest preconditions, and a set of laws [BSC03] and refactorings [Cor04] have been proposed and proved using this semantics.

Another important active research topic is refactoring, whether for object-oriented code or not, aiming to restructuring code to improve quality aspects like, for example, reusability and easy maintenance. In the context of object-oriented languages, Fowler [FBB+99] described a set of refactorings that has become wide-spread among programmers. These descriptions, however, lack a formal guarantee of correctness, as proposed by [Opd92, Utt92, Rob99, TB01] and more recently by [Cor04, BM06]. Refactoring of code is largely available in integrated development environments, but correctness is normally based on test cases executed before and after program refactoring.

The mechanization of formalisations has also been performed. The works of Oheim [vO00] and Huisman [Hui01] are examples of formalisations of object-oriented features using higher-order logic and the theorem provers Isabelle [Pau94, NPW02] and PVS [ORS92]. While [vO00] concentrates on proving Java type-safety, [Hui01] describes and uses a tool LOOP [vdBJ01] for translating Java programs, with or without code annotations, in Isabelle or PVS theories to allow proving properties about these programs.

As already mentioned, the Java Modeling Language (JML) allows introducing specification annotations into Java code. These special annotations describe class, attribute and method invariants which are captured by tools such as LOOP and ESC/JAVA [FLL+02, BRL03] to statically verify programs. These tools, however, are useful when programmers are used to add specifications during the implementation process; for those systems without these annotations, other tools can be used to automatically fill this gap of information in the checked code, but human support is still required. In [NE02] Nimmer and Ernst highlight the importance of these kinds of tools [EPG+06, FL01]. Another practical approach has been the use of Microsoft Visual Studio[Mic07b] to check, during development time, programs written in Spec#, already mentioned.

1.1.2 Unifying Theories of Programming

Sometimes, definitions of object-oriented features using formalisms, descriptions of laws and even mechanization are not enough, and we are requested to compare, or combine, the object-oriented paradigm with others; like, for example, combining object-oriented features with communicating processes, or introducing time in the object-oriented paradigm. This problem can be more systematically overcome if formal descriptions of different paradigms use the same formalism, and, moreover, if it defines a mechanism to integrate these descriptions. These are important capabilities of the UTP framework.

The UTP supports the description of constructs from different paradigms using the same

formalism; each paradigm is characterised by a theory: a set of predicates that describe properties of a particular set of observational variables, and that satisfy particular healthiness conditions, which are filters selecting only valid predicates of a given theory. Combinations of these theories can be used to describe richer paradigms.

In the long term, work in the UTP aims at the creation of a repository of theories that can be compared and combined, in much the same way as software components are used today. In the case of the UTP, however, we are concerned with language semantics, not code. Each new work in this formalism helps to increase this repository of theories, which starts with sequential programming languages, terminating programs (designs), higher-order programming and sequential processes already defined. In this thesis we add a theory of object-orientation (without sharing) to this set and combine it with a theory of pointers presented in [CHW06] and extended in [HCW07].

In [HLL05], we find a description in the UTP of an object-oriented language that handles dynamic binding, pointers and visibility mechanisms, among other object-oriented features; the authors also present a set of rules related to refinement. Another example of an object-oriented language described in the UTP is presented in [QDC03], where the semantics of TCOZ [MD98, MD00], a language that combines processes, classes and time, is defined. The process aspects, however, are the focus of attention.

In [Kas05] Kassios provides a theory of object-orientation in terms of a predicative programming framework [Heh04] similar to the UTP. His objective is to provide general object-oriented concepts independently of the class construct, which most approaches currently consider as a single unit [AL97, MS97, PHM99, CN00, Ral00, PdB03]. He shows, for example, that inheritance and refinement are independent of the class concept, and if some restrictions are imposed to these general concepts then object-orientation is characterised as a particular case.

1.2 Objectives

The long-term goal of the project in which this thesis is embedded is to define a combined UTP theory for reactive, object-oriented designs, and use it to give a semantics to *OhCircus* [CSW05]. This is an object-oriented extension of *Circus* [WC02], a combination of Z [WD96] and CSP [Hoa85, Ros98] whose semantics is based on the UTP. *Circus* is a language designed to allow modeling of concurrent and reactive systems using constructs of Z and CSP together; it also includes a refinement calculus similar to that presented in [Mor94]. A definitive semantics of *Circus* in the UTP is provided in [Oli05]. Our work is an important step to allow the use of a *Circus*-like language to reason about object-oriented programs with reactive behavior.

In this work we:

- ***introduce a theory of object-orientation in the UTP*** that enable us to reason about

relevant concepts of object-oriented programs with an stepwise approach to simplify further integrations;

- ***combine the theory of object-orientation with a theory of pointers*** to allow reasoning about more realistic object-oriented programming languages and exemplify the theories integration process;
- ***describe laws of programming proving their soundness*** to allow behaviour preserving program transformations with, or without, a pointer semantics using an algebraic reasoning as a complement to the inductive approach.

It is important to emphasize that in the UTP the logical background and the concept of refinement are intrinsic to the formalism. As we introduce the constructs of object-oriented programs as a UTP theory we already have a logical framework that enable us to verify if a given program refines a specification. Moreover, in contrast to [AL97, PHM98] the UTP do not distinguish between programs and specifications; they are interchangeable and can be mixed together. Another fact to take into account is that some of the UTP theories have been mechanized [OCW06, Oli05] using the ProofPower [Ltd89] theorem prover, and an extension to cope with high-order procedures and object-oriented features would be desired, but it is not a primary focus at this stage of our research which is concerned with the integration of a theory for object-orientation and a theory of pointers, presented in Chapter 5.

1.2.1 A Brief Example

```
class Account {
    int number;
    float balance;
    //... other attributes
    int getBalance() {
        return this.balance;
    }
    void setBalance(int balance) {
        this.balance = balance;
    }
    //... other set/get methods
    void credit(float value) {
        this.balance = this.balance + value;
    }
    void debit(float value) {
        this.balance =  this.balance - value;
    }
    //... other services
}
```

```
class BAccount extends Account {
    int bonus;
    //...
    void credit(float value) {
        this.bonus = this.bonus + 1;
        super.credit(value);
    }
    void bonify() {
        super.credit(bonus);
        this.bonus = 0;
    }
    //...
}
```

```
class Main {
    public static void main(String[] args} {
        Account x = new BAccount();
        x.credit(100);
        System.out.println(x.getBalance());
        if( x instanceof BAccount ) {
            ((BAccount)x).bonify();
        }
        System.out.println(x.getBalance());
    }
}
```

Figure 1.1: Java like implementation of classes *Account* and *BAccount* and a testing program.

We concentrate on the definition of a theory for object-orientation capable of handling: subtyping, inheritance, dynamic binding, and self-recursive methods. In this subsection we provide an introductory overview of our notation and the features we are dealing with using a short example. Figure 1.1 shows a Java like code for a subset of a simple banking system, where we have a class *Account* and its subclass *BAccount*, which are used by the program *Main*.

The code in class *Main* reveals the importance of a minimum set of features. The class *BAccount* must inherit attributes and methods from *Account*, to allow correct object construction and the selection of the appropriate *credit* implementation (which uses a type casting guarded by a type test) is based on runtime.

In our theory we break down class constructs in separated blocks to independently introduce class names, attributes and methods into an object environment, responsible for recording these information. Each of the approaches discussed in Section 1.1.1 has some kind of environment to record types and other relevant information. We show in Chapter 3 that in the UTP these variables are called observational variables, and in Chapter 4 we define the structure of these variables in our theory. To update these observational variables of our theory we defined constructors for class introduction (**class**), attribute introduction (**att**) and method introduction or redefinition (**meth**). Using our notation, an equivalent representation for the code in Figure 1.1 is presented in Figure 1.2.

We introduce object information using the UTP sequential composition operator (';'). Notice that attribute and method declarations must include the name of the class they belong to; with this we can have an interleaving of declarations; for example all **class** declarations could come before attribute or method. This flexibility simplifies our UTP definitions but pose some restrictions on the declaration order (see Section 4.4).

For the sake of simplicity, and because of their similar treatments, we use only booleans ($\mathbb{B}$) and integers ($\mathbb{Z}$) to represent the primitive types; the identifier **this**, which stands for the current target object is replaced by **self** (mandatory for attribute accesses and updates), and parameters are passed using three different mechanisms[1]: value, result, or value-result. Constructors are parameterless and associated to the expression **new**. All occurrences of **super** are replaced by its macro expansion, and, moreover, since we are dealing with semantics, other simplifications with no loss of expressiveness are considered, as avoiding name clashes for attributes, parameters and methods (except for redefinitions), or trivial transformations like '$((A)x).m()$' to '**var** $y : A;\ y := x;\ y.m();\ x := y$', which eliminates the cast '$(A)x$' by using a fresh local variable y of type A. In the latter case, the command $x := y$ is required only in the absence of pointers.

[1] In Chapter 5 a call-by-reference parameter passing mechanism is also considered.

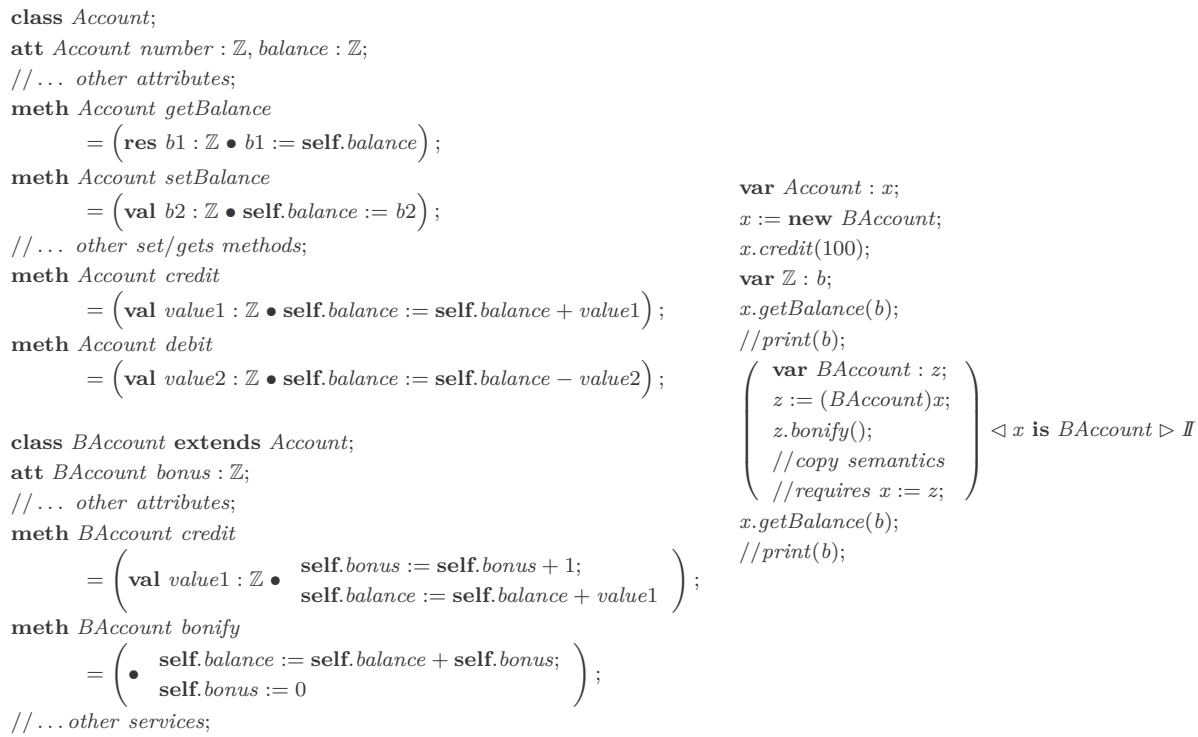

Figure 1.2: UTP representation for classes *Account* and *BAccount* and the testing program.

1.3 Outline

Apart from the introductory chapter, this thesis is organized as follows:

- **Chapter 2** describes previous and related works in more detail than in Section 1.1.1, with a deeper analysis of their characteristics;
- **Chapter 3** presents an introduction to the UTP semantic framework, including an overview of terminating programs, higher-order programming theories, and theories integration that are the basis for our work;
- **Chapter 4** shows the modeling of object-oriented concepts as a theory in the UTP, its variables and behaviours associated to constructs, commands and expressions;
- **Chapter 5** introduces a theory for pointers in the UTP and relates it to the theory of object-orientation presented in Chapter 4;
- **Chapter 6** introduces laws for object-oriented programs that are proved to preserve the semantics;

- **Chapter 7** summarises the results achieved and presents topics for future work.

The Appendixes contain the theories descriptions, proofs for laws and auxiliary results.

Chapter 2

Object-Orientation Formalisation

In this chapter we provide an overview of some efforts in the direction of formalisation of object-oriented languages. We start, in Section 2.1, with some approaches of formalisation and works related to the UTP framework. After that, we consider laws and refactorings (Section 2.2) and mechanization (Section 2.3), concentrating on object-orientation, and finally, in Section 2.4, we present our conclusions.

2.1 Approaches to Semantics

In [AL97], which was revised in [AL03] to include theorem proofs and related works, Abadi and Leino provide a programming logic in the context of object-oriented languages. They describe a logic similar to the Hoare logic [Hoa69] to allow reasoning about partial correctness, pre- and postconditions, in contexts where more sophisticated concepts related to data structures, i.e. inheritance and dynamic binding, are present, as in object-oriented languages. This work represents a starting point for the many others we discuss in this section. In this work well-known concepts of object-oriented programs (like inheritance) are not handled; we show that later extensions of this work include these constructs, notably in an extension by Leino himself [Lei98].

In their approach, objects (its fields and methods) are defined by means of a simple set of constructs, partially borrowed from [AC96]. Primitives for class declaration and inheritance are not provided. Based on these object constructs they define stacks and an object store to represent the runtime environment, modeling their restrictions. Stacks map variable names to booleans or references (sharing is allowed), and stores map object fields to booleans or references, and map methods to their definitions. The constructors of the specification language are fully described in terms of operational semantic rules to show how they alter the program stacks and storage; these rules specify, for example, what happens when a new instance is created. With this operational description, the set of valid programs is now restricted to those which, by mean of rule applications,

reach a final state; that is, those programs for which there exists a finite sequence of derivation steps. Non-terminating programs cannot be handled; attempts to derive a program with an infinite loop diverge.

To create similar instances of an object (class concept is not present), specifying objects with the same attributes and methods, the instance declaration code must be repeated for each instance. Therefore, during object creation one cannot use directly the definition of a type (class) to create an arbitrary instance with its attributes and methods predefined, and then set default values as usual. All attributes and method declarations, including a simulation of dynamic binding, must be written by the programmers. There is no code inheritance.

So far we have discussed how to create arbitrary instances of objects that can related or not: an object-based model. To characterise groups of objects with similar properties and, furthermore, to allow the definition of restrictions in terms of pre- and postconditions for these groups of objects, the concept of *type* has been introduced, and thus a type-based model is defined. The possible types are *Bool* and object types defined by the users. An object type defines which attributes and methods an instance must have to be considered part of a group, usually named *class*. There is a clear distinction between types and instances; they are not explicitly attached.

Inheritance is not explicitly defined in the object type; a type B is considered a subtype of A only if all attribute names and types defined for B are equal, or a conservative extension of those for A, and if all method names are equal, and return types are equal or subtypes of the original methods for A. There is a subtype notion, but not inheritance; notice that the programmers have to rewrite all attributes, and methods redefined or not. It is important, however, to make it clear that redefined methods are not restricted to those which respect behavioural constraints, as those defined in [LW94]. With the introduction of types, a type system for objects can be defined and only programs that are well-formed according to this type system are considered: a close relation between instances and types is established. In particular, an instance introduced in the object store must have its structure defined by a valid type. Operational rules are introduced to show how each of the program constructs can be used in accordance with this type system.

Their final step was to associate pre- and postconditions to specific methods of a given type, the concept of *object specification*. Using a specific set of special functions and variables the expected behaviour of a program in terms of its input and output variables, stacks and storages, can be represented. Using what they call *transition relations*, restrictions to be respected before and after program execution, possibly after some method calls, are defined. There is a parenthood relation between specifications: a specification for a type B can be considered a subspecification of the specification for A if it is more restrictive, that is, all instances that are expected to satisfy specification B must satisfy A. They show that if a valid set of constructs, type definitions, and object specifications, are used, if a final state is reached, and this state satisfy all restrictions it is

indeed the correct one. Using the type and specification systems defined in terms of an operational semantic rules the soundness of the theory is established.

Their logic allows sharing, where all object instances are saved into a shared object store. Recursive types, however, are not allowed. In [Lei98] Leino extended the logic presented in [AL97] to allow recursive types, and introduced single inheritance explicitly. Differently from [AL97], Leino starts presenting the type system of the language and later he defines the specification language. All type system restrictions are described by means of operational semantic rules.

The object environment is formed of a list of declarations of types, fields and methods. The declarations of types are pairs $T <: U$, where T and U are names indicating the subclass and superclass, respectively. It defines explicitly the subtype relation, a partial function from name to name. Two built-in types, *Boolean* and *Object*, are introduced, and judgments to check if a given type belongs to the environment are described. These rules say, for example, that *Object* belongs to the typing environment, and that superclasses must be previously defined. In our theory (see Sections 4.3 and 4.4) we represent these kind of rules as healthiness conditions and restrictions in class declarations.

To introduce an attribute, a function with the name of the attribute is created and a mapping from source class to attribute type is defined. For instance, for an attribute f in a class T with type U, a function $f : T \mapsto U$, where T ranges over objects present in the program environment and U ranges over $T \cup \{Boolean, Object\}$, is created. As with types, all attributes must have different names. The author also introduces the concept of closure of attributes, required by object creation in the presence of inheritance. This closure is common to most formalisation of object-orientation.

Methods are modeled as quadruples $m : T \mapsto U : R$ where: m is the method name, T is the target type (of the object to which the method is applied), U is the result type of the method, and R is a specification relation over the method body, indicating pre- and postconditions, similar to the transition relations in [AL97]. In both approaches [AL97, Lei98], methods are parameterless; objects receive parameters to work with by means of fake attributes set up before calling the method. It does not seem an elegant solution but indeed does not reduce language expressiveness. The forthcoming approaches do not have this limitation. Method names must also be distinct, which conflicts with the concept of dynamic binding, as we explain. The closure of methods returns all methods defined for a given type, but as redefinition of methods is not allowed, dynamic binding is not possible, or even modeled, and, therefore, in this case behavioural inheritance is trivial.

As in [AL97], methods are extended with their specification, contracts described in terms of pre- and post-states of a method. Now, an axiomatic semantics is provided describing what are the valid commands according to the typing system and general relations of pre- and post-conditions. The axiomatic semantics is followed by operational semantic rules defined for the language constructs showing how the stacks and the object storage are affected, as in [AL97]. The

final contribution of this work is a soundness theorem that relates these two semantics, besides introducing a simple manner of introducing recursive types and the subtyping relation. We benefit directly from this strategy to separate blocks of constructs as we show in Section 4.4. In many other approaches [MPH97, Ral00, CN00, QDC03, PdB03, BSCC04, HLL06] the declarations of classes are made in blocks with all their attributes and methods, which make difficult, for example, defining the semantics of a self recursive methods.

In [MPH97], Müller and Poetzsch-Heffter describe how to relate operational semantics of programs to declarative specifications, providing a formal foundation of interface specification. The general idea is to describe abstract data types by means of abstraction functions relating them to concrete implementations. Each interface method of a type has a description of its functional behaviour by means of pre- and postconditions involving logical variables that represent, for example, the state of the system before and after method computation. To make the functional behaviour less abstract, an object model is described with primitive and object types, and its operations, such as assignment to memory locations (references are modeled).

The specification of a method interface is categorised in three parts: functional behaviour, environmental behaviour and sharing properties. The first one is concerned with representation of postconditions in terms of object parameters and abstraction functions based only on the initial state; the second one is concerned with how the object model (runtime) is changed by the method, describing, for example, side-effects of an object creation in the runtime environment; and the third concerns with how shared objects are affected by methods, that is, what are the modifiable objects and how the object instances relationship graph is affected.

Besides method behaviour characterisation, a general form to represent class invariants is presented, describing that for all instances of a given class (and its subclasses) a class invariant must be preserved if the object is alive (allocated and active) and is not null. Abstraction functions are valid only for objects that respect some well-definedness conditions, and thus well-definedness can be considered an invariant property. The pre- and postconditions of methods are now enriched with the class invariants: a Hoare triple $\{P\}m\{Q\}$ used to specify methods or command restrictions, where P is a precondition, m an abstraction present in class C and Q is the postcondition, now is represented by $\{INV_c \wedge P\}m\{INV_c \wedge Q\}$, where INV_c stands for the invariant of class C. The problem of preserving object invariants is a challenging problem itself and has been an active research topic [BDF+04].

To allow behavioural subtyping [LW94] for class inheritance (in program extensions), proof obligations are described. The question to be answered is: how to guarantee that new subclasses preserve the behaviour of the superclass? They use the well-known approach of weakening the precondition or strengthening the postcondition [Mor94]. If the method of the superclass has a pre- and postconditions pair (P, Q), and the subclass has a pair (P_s, Q_s), then $P \Rightarrow P_s$ and

$Q_s \Rightarrow Q$. To prove these implications they use coercion functions to relate attributes and methods of the subclass to the corresponding ones in the superclass. The remaining problem concerns with program extension by introducing classes which are not subclasses of the existing ones; in this case, the invariants of each class must have to be somehow revisited to verify that the new class invariants might not cause any side-effects, in the whole specification.

The work reported in [MPH97] states in a simple and clear manner how pre- and postconditions of methods and class invariants can be specified, and highlights the proof obligations associated to program manipulation or extension. The concepts presented are a starting point to deeper investigation, as explored in the following works of these authors, which we discuss in the sequel.

A work similar to [AL97] is presented in [PHM98], where Poetzsch-Heffter and Müller describe a type system for an object-oriented language, but, unlike [AL97], soundness of the axiomatic semantics is not presented. Features like inheritance and dynamic binding, however, are handled with the objective to reach a formal description closer to practical languages. As in [MPH97], and in contrast to [AL97], the specification related to objects are not embedded into the programs. A language with common object-oriented constructs named KOOL (**k**ernel of **o**bject-**o**riented **l**anguages) is presented. In this language it is possible to represent classes with single inheritance and also abstract classes, which define method signatures (including parameters). Visibility is not mentioned in the work but according to Java standards [GJSB00] the attributes are implicitly protected, and *get* and *set* methods to, respectively, read and update the attributes are included too. As in [AL97] the subclass relation is a partial function with a top most element, here *OBJECT*. *OBJECT* is an abstract class, and there is also two predefined concrete classes *INT* and *BOOL* to represent primitive types.

After the language is defined an object environment is formalised. It describes how objects are linked by references and which ones are considered alive, that is, if the object is present in the environment. Object states (instances) are presented and a set of commands to assign, lookup, check if the object is alive, create a new object and select all objects of a given type, are defined. Axioms relating commands and functions on object states are presented. A Hoare logic is used to describe commands and method behaviour by means of pre- and postconditions. Rules represent the commands and method behaviour using conjunctions, sequent (assumptions), and consequent (resulting effects) clauses. The programming logic involving dynamic binding is also translated as a rule, testing the type of the object and selecting the appropriate method body.

They also present some characterization of open and closed programs. Shortly, open programs are libraries of components that can be reused to write different programs, while closed programs are sets of declarations associated to a single program. A direct impact of this definition is that open programs should ideally have a modular (compositional) proof strategy to allow increase functionality without the need of revalidating all already proven results; this is a topic of discussion

in [MPH97]. For closed programs, where all software components are well-known (already selected), the proofs can became easier, although open programs are more important to achieve reusability, and in this case the proofs are lengthier. In Chapter 6 we present some laws of object-orientation that are valid not only for closed programs, but also for libraries.

Two verification strategies are described in [PHM98], one top-down, that can be summarised in three steps: (i) define specifications for methods of abstract types; (ii) define specifications for the concrete, or abstract, subtypes and prove they are compatible; and (iii) prove that the method implementations in concrete classes satisfy their specifications. The bottom-up strategy starts from the implementations and using subsumption rules, the proof obligations are derived. In the latter case, in the context of closed programs, some preconditions of the rules can be eliminated, simplifying the verification process. The language also allows attribute overriding. As we do not believe that this feature increases object-oriented elegancy, and rather complicates too much the program and its semantics, our theory does not allow this kind of redefinitions, but in other approaches as [Ral00, PdB03], and Java itself, it is allowed. The access to attributes or to superclass methods in [PHM98] is done using a different notation ('@'), while in Java no extra information is required (always '.'), resulting, sometimes, in programmatic errors, specially with the less attempt programmers.

This work includes important features of object-orientation closer to real object-oriented languages, but as said before it does not provide a soundness proof as its correlated work [AL97]. This limitation was overcome in the next work of these authors. In [PHM99], Poetzsch-Heffter and Müller extended [PHM98] to cope with encapsulation. The KOOL language was replaced by a simpler subset of Java named Java-K. A structural operational semantics was used to describe rules in this language. The programming logic is similar to the previous work, although a soundness theorem was presented. The operational semantic rules were mapped to a higher-order logic and proved manually.

In [Ral00, BMvW00], extensions of [MS97], Back *et al.* build a logical framework to reason about object-oriented programs; this framework is a conservative extension of refinement calculi [Mor94, BvW98]. The focus of the work is to allow object substitutability, also referred to as polymorphism, in client applications by refining the object class definitions. A client application is defined as a non-deterministic choice of method calls of an object; they show that for a class C', which refines C, substituting instances of C by instances of C' in the clients that use C is refinement. They argue that dealing only with syntactic aspects of subtyping, such as method signatures, is a decidable problem, while behavioural-inheritance, discussed in works like [Ame90, LW94, Mey97], is not, and thus could perhaps be verified by theorem prover like HOL [GM93] or PVS [ORS92], but none mechanization is provided. The same strategy to translate object constructs to a higher-order logic was used in [PHM99] and specially in [vON99], where

the semantics for a subset of Java is described to prove Java type-safety.

In the work of Back *et al.* all attributes are private (hidden), all methods are public, and an object instance is a tuple with attribute values and the object type. They do not allow recursive types, but allow pointers. Subclasses can redefine attributes and methods. Methods of the superclasses can be called using the 'super' special reference. Dynamic binding is modeled as an angelic choice between subclass and superclass method definitions; the choices are guarded by assertion testing the type of the instance, if the type test is false the alternative behaves like abort ($\bot$, an unpredictable program) and other alternative can be chosen (semantics of angelic non-determinism); only one of the alternatives is selected at a time. If more than one guard is valid, a non-deterministic choice is made. These kinds of tests based on object type are common to some approaches [BSCC04, MPH97], including ours. They also handle behavioural aspects of the substitution, defining that, for a given instance of a subclass to be in conformance with the superclass behaviour, newly introduced methods must not change attribute values, or change the superclass attributes in the same way as superclasses used to do. This can be formally characterised as in [MPH97], using coercion functions.

Another effort in the formalisation of object-orientation is the Refinement Object-Oriented Language (ROOL) [CN00] project; ROOL is a subset of sequential Java with visibility, dynamic binding and copy semantics. The refinements are defined for commands and parametrised commands [Bac87] in an object-oriented context. In [CN00], Cavalcanti and Naumann provide a complete description of the language constructs, defining the semantics of the language in terms of weakest preconditions.

A typing system is defined and rules related to types are introduced. A semantics for programs in ROOL is defined by induction on the typing rules; properties are enumerated and proved. The approach, however, leaves out features such as mutually recursive methods. The main weakness, however, is the need for inductive proofs, which are not conducive to reuse and extension. In our thesis the focus in a general theory of object-orientation and proofs are made straightforward at the semantic level, thus the results apply to any object-oriented language.

In the Unifying Theories of Programming (UTP) [HH98], Hoare and He establish a framework to allow comparing and reasoning about different programming paradigms using a relational calculus; predicates over an alphabet of observational variables are used to specify relations in the style of VDM [Jon90] and Z [WD96]. The description of a theory for object-orientation is important to allow the integration of different theories such as sequential communicating processes [Hoa85, Ros98] using the same formalism. In the next chapter we introduce basic concepts of UTP required to understand our theory, and its integration to a theory of pointers presented in [CHW06]. A detailed description of this framework is presented in [HH98] and a shorter alternative can be found in [WC04]. In the following, we discuss works related to object-orientation in

the UTP and other predicative styles.

In [HLL05, HLL06] He *et al.* present a syntax and semantics for an object-oriented language in the context of the UTP [HH98], named Refinement Calculus for Object Systems (rCOS). They define a language similar to ROOL [CN00], but with a reference semantics. Some basic concepts of the UTP, such as designs, are revisited, and refinement of designs is described. The static and dynamic semantics are described in detail. Observational variables are introduced to record object references and instances, as the previous approaches, with the exception of ROOL, which has a copy semantics. A surprising feature, perhaps, are the well-definedness rules for class definition, which allow two completely separated inheritance lines to be defined. It is not true that 'Object' is the top most element of the transitive and reflexive subtyping relation. This is not a problem to the other definitions presented in the work, but it is, at least, unusual and makes it impossible to have a variable which can assume all possible object runtime values. In our approach, for example, this singularity would complicate the semantics of method call, where the 'self' variable must have the dynamic type of the target object, and as we handle all method calls uniformly, the unique valid type would be 'Object'.

rCOS works with closed programs, where a sequence of classes is followed by a program, that is, $Cdecls \bullet Main$. The refinements of class declarations and programs are defined. These refinements are grouped in: *system refinements*, where a program $Cdecls \bullet main_1$ is declared a refinement of $Cdecls \bullet main_2$ if for all external observational variables the meaning of $main_1$ implies the meaning of $main_2$ (denoted by $[\![main_1]\!] \sqsupseteq_{sys} [\![main_2]\!]$); and *structural refinements* where the class structure is allowed to change, including or removing classes, attributes or methods; in this case the following implication must hold $\forall\, main \cdot (Cdecls_1 \bullet main \sqsupseteq_{sys} Cdecls_2 \bullet main)$. The set of class declarations $Cdecls_1$ is considered a refinement of $Cdecls_2$. For example, the change of a class visibility from public to private, once this class is never used in *main*, is an example of system refinement. For structural refinement, to perform a valid refinement we cannot remove any public attribute or method used in the main body, for example. This restriction is essential in the context of component repositories, if we do not know which other components are using a given service we definitely cannot change component signatures.

The definition of a program is the reference for refinement theorems for upward and downward simulation, and their proofs. Basic laws of structural refinements, similar to those already presented in [BSCC04], and for patterns like *responsibility assignment*, *data encapsulation*, and *high coercion* are defined. The big goal of their work is to provide a language to study structural and further behavioural refinements that can help to introduce formalism in UML diagrams.

In [MD98] Mahony and Dong present a formal notation named Timed Communicating Object-Z (TCOZ), which is a combination of Object-Z [Smi00] and Timed CSP [DS95, SH02]. In [QDC03] Qin *et al.* describe the semantics of some features such as inheritance, encapsulation, dynamic

binding, and timing constructs using the UTP. The syntax of the language is presented with all observational variables of the theory. Using these variables, they describe the semantics of classes and process constructions. The language is not fully described in the UTP; they concentrate on process aspects of the languages that can be easily addressed by the UTP formalism, and the main focus of the work is not the object-orientation concepts, but rather the process behaviour.

In [Kas05] Kassios introduces a theory to decouple concepts of object-orientation that are traditionally embedded in the idea of class, such as inheritance and encapsulation, using the predicative programming style defined in [Heh04], very similar to the UTP. While traditional formal approaches for object-orientation [AL97, MS97, PHM99, CN00, Ral00, PdB03] introduce all these features directly in a declaration of a class. Kassios' idea is to provide general descriptions of object specifications, refinements and inheritance, among others, independently of the class construct and of each other, thus reaching the decoupling. This results in very general specification constructs, of which those usually found in object-oriented languages are a special case. We also adopt the decoupling of concepts and constructs in our theory to allow their combinations with other concepts or theories, in the spirit of the UTP framework.

2.2 Laws and Refactorings

Another important aspect of a software is the fact that a given functionality can be implemented by infinite programs, and there is no doubt that some are better than others in terms of performance or resources requirements. Sometimes, not rarely, developers are requested to increase programs overall performance, or even rewrite them to increase readability and reusability to facilitate further maintenance or extension. In these situations, a big challenge is to guarantee (and verify) that the new updated system behave exactly like the previous one, or better. The matter of readability, or even reusability, are sensitive to controversy, but the result of programs must be the same for both contexts. A common alternative to the preservation of behaviour is the construction of test case suits that are executed against programs before the transformations (refactorings) and after; if the results are equivalent the new revised program is considered correct. Unfortunately, as is well-known, testing can only show the presence of errors, but not their absence. A more rigorous alternative is to prove that the original system behaviour corresponds to the refactored one.

There are many works that handle this problem of changing programs at different levels of abstractions, from a completely informal approach based on test cases [FBB+99], to more formal ones [Opd92, Utt92, Rob99, TB01, Cor04, BM06]. In this thesis we consider the work presented in [Cor04] as a comparative source for the laws and their proofs, due to its close connection to our work, including (in the first moment) the adopted copy semantics, also present in our object-oriented theory.

In [BSC03] Borba *et al.* enumerate algebraic laws for ROOL programs that were the inspiration for our laws (in Chapter 6), but they consider only closed programs; the results do not apply necessarily to open systems (libraries). The provisos of the laws describe the conditions that need to hold for changes in the programs to preserve behaviour. Laws for classes, attributes, methods and commands are proposed and described. In order to postulate the relative completeness of the proposed laws, a normal form for object-oriented programs in ROOL is defined and a sequential list of steps required to reach that normal form is presented. Each step relies on the application of some law. In [Cor04], Cornélio describes and proves refactorings in ROOL. A subset of the refactorings presented in [FBB+99] is selected and described using the ROOL syntax. All laws presented in [BSCC04] are proved by induction and used in the proofs of more complex refactorings. Due to the copy semantics, not all refactorings of [FBB+99] are handled.

In Chapter 4 we also use a copy semantics. In Chapter 5, however, we combine the object-orientation theory with a theory of pointers, and then in Chapter 6 we verify the applicability of the laws in both contexts.

2.3 Mechanization

Some works have been done on the mechanization of object-oriented languages, most of them in the end of the 90's, concentrating on the Java language. An example of a complete formalisation of a subset of Java is [vO00], encoded by mean of a higher-order logic, mechanised using the Isabelle [Pau94, NPW02] theorem prover to check type-safety, soundness and relative completeness for an object-oriented version of the Hoare logic (previously described in [NvO98]).

Huisman [Hui01] presents a semantics for Java in higher-order logic and describes a tool, named LOOP [vdBJ01], which translates the programs to input theories for Isabelle or PVS provers in order to check properties of the programs, for example, if a method return value is not null.

In the direction of bringing formal specifications closer to the programmer's world, simplifying the introduction of formal methods in current programming practices, there is the example of Java Modeling Language (JML) [LBR99, BCC+03] which aims to introduce annotations in Java code to specify object invariants and pre- and postconditions of methods, among other constructs, as model fields[1]. These annotations are turned into run-time verifications embedded in the bytecode generated by the JML compiler.

The most advocated benefit of JML is that its notation uses the host language, Java, to introduce specifications into code; thus, being attractive to program developers. Moreover, the use of JML does not impose a fixed design methodology to its users, (in contrast to B [Abr96]); specifications can be inserted at any stage of the development according to programmers needs.

[1] Variables present at the specification level.

Other important aspect of using JML is the fact that there is a growing set of tools [vdBJ01, FLL+02, BRL03] converging to use its notation, enabling the developer with a rich toolkit to be used in different contexts.

One example of JML tool is the Extended Static Checker for Java (ESC/Java) [FLL+02] which checks statically the conformance of Java code with its annotations. This tool, however, as the authors highlight, is not comitted to guarantee soundness or completeness. Depending on developer demands another tool must be used to check the programs, but it represents one step ahead in software verification.

Even after mechanization progresses, there are other practical aspects to be handled, like legacy systems; more specifically, on how to provide ways to automatically introduce specifications into code, as highlighted by Nimmer and Ernst in [NE02]. The use of inference tools like Daikon [EPG+06] and Houdini [FL01] can reduce the time consuming task of review all legacy system code to introduce, sometimes, well-known restrictions. Unfortunately, as usual, some specifications require human interaction. These tools, however, reduce considerably this need [BCC+03].

There is also a methodology to introduce formal descriptions into code for Spec# [BDF+04, BLS05]; this initiative is called Boogie [BCD+06], which aims to introduce formalism in the .NET platform, and the tool support is integrated with the Microsoft development tool Visual Studio[Mic07b]. Up to our knowledge, this seams to be the best integration of formal verification in a real widespread-use development tool today.

Its is important to remark that there is a trade off between factors that determine the selection and the use of mechanization tools such as: the completeness of the object verification; the expressiveness of the specification language; time and cost budgets. It seems reasonable to use different tools depending on these restrictions. In the case of Java, the convergence of the specification language to JML has helped to enrich developer toolkits for different needs.

The refactorings, in general defined as laws of programming [Hoa69, HHJ+92, TB01] or informal descriptions [FBB+99], are supported by many, commercial [Bor07, Mic07b] or non-commercial as [The07], programming tools for different languages. The refactorings are performed automatically, but the correctness of such transformations is checked by informal verification techniques, such as compilation and test cases.

2.4 Conclusions

In this chapter we have briefly presented some approaches and formalisms used to model object-oriented paradigm features and its relations, whenever possible establishing relations with the work presented in this thesis.

We have seen that Abadi and Leino started with a relatively simple model of objects, and it

evolved to a formalism where Leino introduced recursive types. In both cases soundness of the theory was considered and stated in terms of an operational semantics. In the latter approach Leino highlights, however, that methods whose return types are the same as the owner class cannot be defined.

Müller and Poetzsch-Heffter defined a language with constructs close to real programming languages. They began defining some general techniques for object-oriented programs, then a more detailed language was introduced to handle features as inheritance and dynamic binding, but no soundness proof is provided. Later, they extend the work to cope with encapsulation, and defined the object-oriented concepts using a higher-order logic that allowed a manual proof of soundness.

Next, the approach by Back *et al.* provided a characterisation of refinement in the point of view of client applications, which are modeled as a non-deterministic choice of object methods. In this approach dynamic binding is modeled as an angelic choice.

With a strong refinement background there is ROOL, by Cavalcanti and Naumann, a language that, besides the basic object-oriented constructs, contains refinement calculus constructs, as specification statements. Its semantics was defined in terms of weakest preconditions, and laws for refactorings in ROOL are described by Borba *et al.*, and Cornélio.

The work of He *et al.* is the first to consider the introduction of object-orientation in the UTP. They extend the theory of designs and provide a set of observational variables to record types, attributes and methods information. Unlike their approach and the ROOL, which have to calculate a method meaning every time it is requested, our theory uses higher-order programming to record method meanings during the environment loading time. Method meanings are recorded in special variables, and when necessary they are directly accessed, but we pay the price that the order of declarations becomes relevant.

The formulation and formalisation of refactoring has been recently emphasised due to Fowler's book, and a growing set of formalisms and tools have been described and developed to mechanize these program transformations. In this thesis we provide some laws and prove their soundness as presented by Cornélio, but with a significant difference that we handle a general theory rather than syntax for any specify language.

Finally we have seen that mechanization of object-oriented features using different languages and tools is possible, but we are still a little far from the goal of having a fully integrated tool to allow development, verification and refactoring of object-oriented programs. In fact the spreading of JML, for Java, is one step ahead to achieve it. In the following chapters we give our contributions to creating a big repository of theories using the same formalism, the UTP. Perhaps, not in a distant future, tools for UTP will benefit from all designed theories, and further, support the task of creating new hybrid paradigms of programming.

Chapter 3

Introduction to the Unifying Theories of Programming

This chapter is dedicated to a brief introduction to the concepts of the UTP required to understand the theories presented in the following chapters. In Section 3.1 we enumerate some basic concepts and constructs of the UTP. Next, in Section 3.2, we reproduce some valid laws in the UTP framework and in Section 3.3 the concept of refinement in the UTP is presented, with its lattice correlation. In Section 3.4 the theory of terminating programs is described and some basic constructs of the UTP are revisited to reflect new restrictions of terminating programs. Then, in Section 3.5, another form of characterising valid predicates in a theory is shown and, in Section 3.6, the possibility of using programs as values of variables is considered: the higher-order programming theory. In Section 3.7 we introduce the concept of Galois connections to integrate theories and in Section 3.8 we define closeness. Finally, in Section 3.9 we provide a short overview of the thesis context.

3.1 Introduction

The UTP is based on alphabetised predicates that describe possible values of observational variables that record relevant information about the programs of interest. In this way, we specify programming constructs of different paradigms using a common formalism.

Theories are characterised by a set of observational variables and by healthiness conditions. For instance, in time-aware systems we can use a variable to record time information (*clock*); in communicating systems we observe the traces of events occurred (tr). The set of observational variables of a theory is called the alphabet. The predicates of a theory refer only to variables in the alphabet, and given a predicate P we write $\alpha(P)$ to refer to its alphabet. The free variables

of a predicate must belong to its alphabet.

When the alphabet is composed of undecorated names, which are used to represent the initial value of the corresponding observational variables, and dashed names that represent the values of the variables in a later or final observation, then the predicates define relations. For example, $clock' > clock$, where the decorated variable (‘′’) represents the final value of *clock*. In this case, we say that the alphabet contains input and output variables; the input set is denoted by $in\alpha(P)$ and the output by $out\alpha(P)$. A relation where the output alphabet is equal to the input alphabet (with decorated versions of the variables) is a homogeneous relation, $in\alpha(P)' = out\alpha(P)$. As in [HH98] we use letters like: P,Q or R to represent predicates; b,c,d to represent conditions[1]; and x,y,z to represent variables.

An example of a relation is provided by the definition of an assignment $x := e$ of an expression e to a variable x, in the context of a theory of general relations, in which the only variables of the alphabet are the programming variables, and their dashed counterparts.

$$x := e \mathrel{\widehat{=}} x' = e \land y' = y \land z' = z \ldots$$

In this example, we take the input alphabet $in\alpha(x := e)$ to be $\{x, y, z, \ldots\}$, and the output alphabet $out\alpha(x := e)$ to be $\{x', y', z', \ldots\}$. In fact, the alphabet of an assignment has to be explicitly defined, but for simplicity it is often omitted. The definition states that the final value of x is e and all other variables of the alphabet have their values preserved. A degenerate form of assignment useful for reasoning about programs is $I\!I$, named skip. It is an empty assignment where all variables have their final values preserved.

$$I\!I \mathrel{\widehat{=}} x' = x \land y' = y \land z' = z \ldots, \textbf{ where } \alpha(I\!I) = \{x, x', y, y', z, z', \ldots\}.$$

Sequential composition, denoted by $P;\ Q$, is defined by an existential quantification that relates intermediate values of the variables. The final values of the variables as defined by P are taken as the initial values of the variables by Q. In this case, $out\alpha(P)$ is required to be equal to $in\alpha(Q)$, after each of its variables are dashed. These intermediate values are represented by w_0 in the definition of sequence.

$$P;\ Q \mathrel{\widehat{=}} \exists\, w_0 \bullet P[w_0/w'] \land Q[w_0/w], \textbf{ where } out\alpha(P) = in\alpha(Q)' = w'$$

The declaration and undeclaration of variables are defined separately, also in terms of existential quantifications.

$$\textbf{var } x \mathrel{\widehat{=}} \exists\, x \bullet I\!I \ , \textbf{ where } \alpha(\textbf{var } x) = \alpha(I\!I) \setminus \{x\}$$
$$\textbf{end } x \mathrel{\widehat{=}} \exists\, x' \bullet I\!I \ , \textbf{ where } \alpha(\textbf{end } x) = \alpha(I\!I) \setminus \{x'\}$$

[1] Predicates which perform tests on initial variable values.

This separation is useful in laws of programming related to variables. An important construction involving variables is that of variable blocks.

$$\textbf{var } x;\ Q;\ \textbf{end } x \mathrel{\widehat{=}} \exists\, x, x' \bullet Q$$

A conditional $P \lhd b \rhd Q$ is defined as a disjunction of conjunctions. If the condition b is valid, then the behaviour is that of P, otherwise it is that of Q, provided the alphabet of the condition b is a subset of the alphabet of the conditional branches P and Q. Hoare and He use an infix notation that is helpful in the definition of laws of programming.

$$P \lhd b \rhd Q \mathrel{\widehat{=}} (b \wedge P) \vee (\neg b \wedge Q), \textbf{ where } \alpha(b) \subseteq \alpha(P) = \alpha(Q).$$

Recursive definitions are provided as fixed points $\mu\, X \bullet F(X)$, where F is a function from predicates to predicates. One example is the while loop, denoted by $b * P$. While the condition b is valid the behaviour is that described by P, otherwise the loop finishes. The semantics of while is defined in terms of a recursion in the usual way.

$$b * P \mathrel{\widehat{=}} \mu\, X \bullet P;\ X \lhd b \rhd I\!I.$$

The non-deterministic choice $P \sqcap Q$ is defined as a disjunction, and both alternatives must have the same alphabet.

$$P \sqcap Q \mathrel{\widehat{=}} P \vee Q, \textbf{ where } \alpha(P) = \alpha(Q).$$

These are some of the UTP basic constructs. As expected, each one is mapped to a predicate over a specific set of variables, an alphabet. Now we show some laws that are derived directly from these definitions.

3.2 Laws

In [HH98], laws of programming are introduced progressively as the set of language concepts is extended. As an example, for conditional, a subset of the laws presented in [HH98] is reproduced below; the names (labels for the laws) used in [HH98] are adopted here as well.

L1 of conditionals: $P \lhd b \rhd P = P$	*idempotency*
L2 of conditionals: $P \lhd b \rhd Q = Q \lhd \neg b \rhd P$	*symmetry*
L3 of conditionals: $(P \lhd b \rhd Q) \lhd c \rhd R = P \lhd b \wedge c \rhd (Q \lhd c \rhd R)$	*associativity*
L4 of conditionals: $P \lhd b \rhd (P \lhd c \rhd Q) = (P \lhd b \rhd Q) \lhd c \rhd (P \lhd b \rhd R)$	*distributivity*
L5 of conditionals: $P \lhd \textbf{true} \rhd Q = P = Q \lhd \textbf{false} \rhd P$	*unit*
L6 of conditionals: $P \lhd b \rhd (Q \lhd b \rhd R) = P \lhd b \rhd R$	*unreachable-branch*

L7 of conditionals: $P \lhd b \rhd (P \lhd c \rhd Q) = P \lhd b \vee c \rhd Q$ *disjunction*

These laws are valid for all conditions and predicates, as long as their alphabets obey the side-conditions required by the definition of the conditional. Basic laws can be proved by direct translation to logical predicates, like the proof of **L1** below, and more complex laws can be derived from the basic ones.

L1 : $P \lhd b \rhd P = P$

Proof.

LHS	*[conditional definition]*
$= (b \wedge P) \vee (\neg b \wedge P)$	*[case analysis on b]*
$= ((\mathbf{true} \wedge P) \vee (\neg \mathbf{true} \wedge P)) \vee ((\mathbf{false} \wedge P) \vee (\neg \mathbf{false} \wedge P))$	*[propositional calculus]*
$= (P \vee \mathbf{false}) \vee (\mathbf{false} \vee P)$	*[propositional calculus]*
$= RHS$	□

There are laws for most part of the constructs, and one of the laws presented for non-determinism is that of disjunction distributivity, reproduced below. Other laws can be introduced for each constructor of a new theory defined in the UTP.

L3 of relations: $(\sqcap S) \sqcup Q = \sqcap\{P \sqcup Q \mid P \in S\}$, **where** $P \sqcup Q \mathrel{\widehat{=}} P \wedge Q$.

Labels for laws can be reused, the former **L3** is defined for conditionals only, and the remainder is a law for general relations. In the text we make clear which one we are using whenever necessary.

3.3 Refinement

Hoare and He have also shown that the set of alphabetised predicates form a complete lattice with the ordering defined by universal implication ($\Rightarrow$). This is a very important result, since different paradigms can be mapped to alphabetized predicates there is a general notion of refinement which is common to them all.

They establish that a program P refines a program Q if, and only if, $P \Rightarrow Q$, for all possible values of the variables of the alphabet: $\forall\, x, x', y, y', \ldots \bullet P \Rightarrow Q$. This is denoted shortly by the universal quantifier $[P \Rightarrow Q]$, and the notation $Q \sqsubseteq P$ is used to represent that Q is refined by P. In all theories of the UTP, the refinement relation is characterised in the same way. The dual symbol $\sqsupseteq$ can also be used and $P \sqsupseteq Q$ is read as 'P refines Q'.

The bottom of this lattice is denoted by $\bot$, also named abort. Formally we have that any program refines abort, $\bot \sqsubseteq P$; this follows directly from its definition: $\bot \mathrel{\widehat{=}} \mathbf{true}$, that is, for any program we have that $P \Rightarrow \mathbf{true}$. On the opposite side, we have the top of the lattice $\top$, named miracle, formally defined as $\top \mathrel{\widehat{=}} \mathbf{false}$; for any P we have $P \sqsubseteq \top$, that is, $\mathbf{false} \Rightarrow P$ (vacuously better).

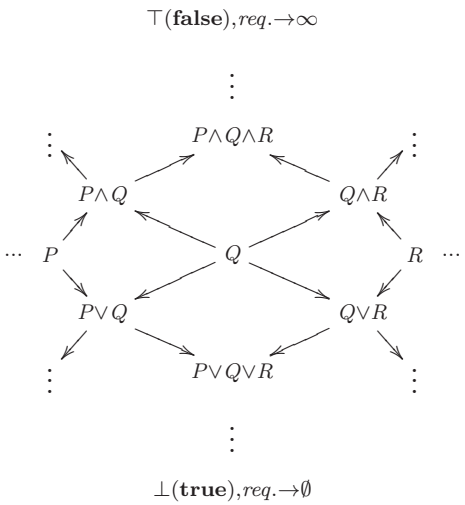

Figure 3.1: Lattice of predicates.

Informally, in Figure 3.1 we have abort as a general program that satisfy a set of disjunctive requirements (possibly empty, $requirements \to \emptyset$) and in the opposite side we have miracle that is infinitely restrictive (infinite set of conjunctive requirements, $requirements \to \infty$). Abort is useless because its behaviour is completely unpredictable, and miracle is simply impossible to implement, but they are useful in reasoning about programs. The set of practical useful programs lies between these two extremes.

Example 1 (Refinement)**.** If we have a specification $clock' > clock$, we are requested to write a program that increases the clock value. There is an infinite set of programs which satisfy this specification, including miracle. Consider a very simple one '$clock := clock + 1$', this is indeed a refinement of the specification as the proof below verifies.

$clock' > clock \sqsubseteq clock := clock + 1$ *[⊑definition]*
$= clock := clock + 1 \Rightarrow clock' > clock$ *[definition of assignment]*
$= clock' = clock + 1 \Rightarrow clock' > clock$ *[numbers properties (n+1>n)]*
$= clock' > clock \Rightarrow clock' > clock$ *[propositional calculus]*
$= \mathbf{true}$ □

This is a simple example of how to prove refinement in the UTP.

3.4 Terminating Programs

The theory of relations is not appropriate to reason about termination. This limitation becomes clear when we calculate the meaning of the program '**true**; $x := e$', surprisingly it has the same

meaning of '$x := e$'.

$\mathbf{true};\ x := e$ *[definition of composition and assignment]*

$\exists\, w_0 \bullet \mathbf{true}[w_0/w'] \wedge (x' = e)[w_0/w]$ *[propositional calculus, substitution, w_0 is not free in e]*

$x' = e$ *[definition of assignment]*

$x := e$ □

That is, a program that after an abortion still behaves like a valid assignment, which is a contradiction. For that end, Hoare and He define a subset of this theory comprising the predicates that can be written as pairs of pre- and postconditions, and have special observational variables ok and ok' to record whether the program has started and ended successfully. The predicates of this new theory are called designs. The definition of a design is presented below, where P stands for the precondition and Q for the postcondition of the design.

$$P \vdash Q \mathrel{\widehat{=}} ok \wedge P \Rightarrow ok' \wedge Q$$

If the design starts, and its precondition holds, then it is certain to terminate and to establish its postcondition; otherwise, no restrictions are enforced. Graphically we could represent the relationship between the set of predicates, relations and designs as presented in Figure 3.2, where all sets are infinite. The designs pose restrictions on variables ok and ok', we will see that these restrictions are modeled also as healthiness conditions.

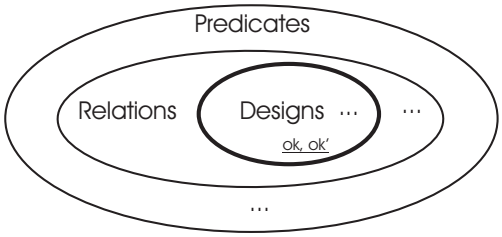

Figure 3.2: Designs subset.

In the theory of designs, assignment and skip have different definitions. Also, the theory of designs is a complete lattice, but its bottom and its top are different. The assignments, and other programs, now have to consider a precondition to be valid. For the sake o simplicity the definition assumes that expression e is well-defined and yields a value compatible with that of x. In the basic UTP, variables and results of expressions are not explicitly typed, in Chapter 4 we constrain it. The new definitions are as follows.

$x := e \mathrel{\widehat{=}} \mathbf{true} \vdash x' = e \wedge y' = y \ldots$

$I\!I_d \mathrel{\widehat{=}} \mathbf{true} \vdash x' = x \wedge y' = y \ldots$

$\bot_d \mathrel{\widehat{=}} \mathbf{true} \vdash \mathbf{false}$

$\top_d \mathrel{\widehat{=}} \mathbf{false} \vdash \mathbf{true}$

In [WC04] the law below is presented for a sequential composition of designs in which the pre-condition of the first design is a condition. We use it to simplify our proofs of object-oriented laws.

T3':$((p_1 \vdash Q_1);\ (P_2 \vdash Q_2)) = (p_1 \wedge (Q_1\ wp\ P_2)) \vdash (Q_1;\ Q_2)$
where p_1 is a condition, **and** $Q_1\ wp\ P_2 = \neg(Q_1;\ \neg P_2)$

This is a specialisation of a law that applies to any sequence of designs presented in [HH98].

We also define a special form of design $\{b\}_\perp$ called an assertion; if b holds, this design skips, otherwise it aborts.

$$\{b\}_\perp \mathrel{\widehat{=}} I\!I_d \lhd b \rhd \perp_d$$

This construct is useful for reasoning.

3.5 Healthiness Conditions

An alternative way of characterising the set of predicates of a theory is using what Hoare and He called healthiness conditions (HC). They impose restrictions on the set of predicates that can be part of a theory. For example, only those predicates that satisfy the conditions **H1-2** of Table 3.1 are considered designs. That is, we can write predicates that satisfy these conditions as implications involving ok and ok'.

Name	Definition	Description
H1	$P = (ok \Rightarrow P)$	Program start.
H2	$P[\mathbf{false}/ok'] \Rightarrow P[\mathbf{true}/ok']$	Non-termination cannot be required.

Table 3.1: Healthiness conditions for designs.

In other words, the subset of designs represented in Figure 3.2 is filtered from relations by using these HCs. An important characteristic of HCs is that they are defined as idempotent functions. Once we apply a HC to make a predicate healthy, there is no point in applying this HC again. Just one application of a given HC to a predicate is enough to determine its validity or not.

3.6 Higher Order Programming

Another theory introduced by Hoare and He is that of higher-order procedures [Nau95]. Variables are now allowed to be bound to abstractions, like, functions or procedures, which are them values. For example, we can have a variable p with a conditional $r := \mathbf{true} \lhd x\%2 = 0 \rhd r := \mathbf{false}$ as its value, an abstraction that tests if a given variable x is even and records this information in a

variable r. In this case the value of p is the text of the program. The inclusion of parameters in procedures is allowed and different interpretations of λ-abstractions define the types of parameter passing mechanism.

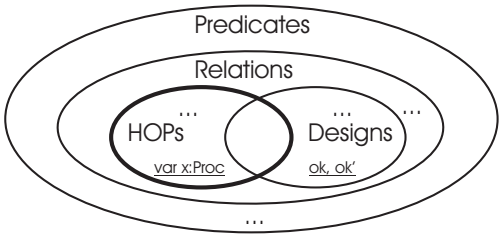

Figure 3.3: Higher order procedures subset.

In Figure 3.3 there is an infinite subset of relations characterised by such types of variables (HOPs). Notice that this set has an intersection with designs, that is, designs where variables can have higher order procedures as values.

In Chapter 4 we show that our theory can be characterised in terms of designs, and that it includes observational variables whose values are program texts; in this way we combine the theories of designs and higher-order procedures. In fact we introduce some extra observational variables to record object-orientation information and restrict this theory to that subset where designs and higher order procedures lies.

3.7 Theories Integration

We have commented that theories in the UTP have been, progressively, developed and these theories are expected to be combined and interact with another. This is the topic of this section.

In some cases, theory integration is a very simple task; for example, one theory can be a subset of another. In these cases, for example, a function that maps all elements of the smaller to the larger one and vice-versa (restricted to a subset) can be defined. A general form of characterizing this mapping is a function L from $\mathbf{S}$ to $\mathbf{T}$, where $\mathbf{S}$ is the set of predicates of a theory and $\mathbf{T}$ is the set of predicates of the other one. If such function exists we say that there is a *link* between them.

The Figure 3.4 (a) is a diagrammatic representation of links like these. Notice, however, that if $\mathbf{T}$ is a more general theory such link associates the elements of $\mathbf{S}$ to its projection into $\mathbf{T}$, which we refer as $\underline{\mathbf{S}}$. If all elements from one theory can be mapped into the other and vice-versa these theories are said to have the same expressivity power. In this case, however, they are too similar that they integration becomes uninteresting.

In fact, the whole point of theories integration is how to find a link between them, that is,

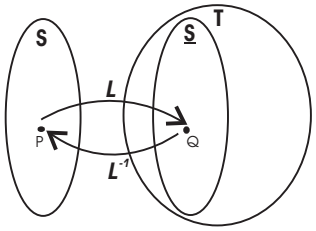

Figure 3.4: A subset theory relationship.

find mappings that translate a predicate from one theory to its corresponding predicate in the other. This is usually made by functions; in the subset example, if the L function exists, and P and Q are member of **S** and **Q**, we have an inverse L^{-1} and properties like $L^{-1}(L(P)) = P$ and $L(L^{-1}(Q)) = Q$ holds. Moreover, in theories with the same signatures[2], the L is trivially the identity function $id : \mathbf{S} \mapsto \mathbf{T}$, and its inverse is $id^{-1} : \mathbf{T} \mapsto \mathbf{S}$ with the domain restricted to those elements in **S**. The problem of signature has also to be handled, but for simplicity we assume that the same symbol for different theories denote the same behaviour, unless clearly stated.

The subset case is just one of the possibilities of theories relationship, we have also to consider two theories that are not necessarily related by subset relationship. More specifically, a theory can describe more things (predicates) than the other is capable of handling, and vice-versa. Therefore, we cannot expect that the function L from **S** to **T** admits an inverse. However, they have parts in common. That is, probably we need one function L to map from **S** to **T**, and another function R from **T** to **S**, which are usually named left and right adjoint. This kind of link is named a Galois connection, whose definition we borrow from [HH98] and adapt to our representation in Figure 3.5.

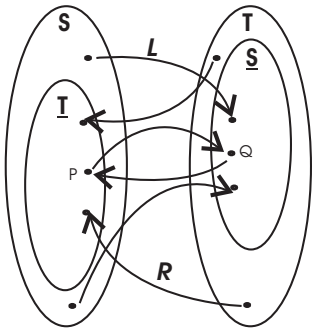

Figure 3.5: A Galois connection.

Definition 1 (Galois connection)**.** Let **S** and **T** be complete lattices. Let L be a function from **S** to **T**, and let R be a function from **T** to **S**. The pair (L,R) is a Galois connection is for all $P \in \mathbf{S}$

[2]Symbols used in a theory and their semantics.

and $Q \in \mathbf{T}$

$$L(P) \sqsupseteq Q \text{ iff } P \sqsupseteq R(Q)$$

R is called a weak inverse of L, and L is called a strong inverse of R.

In [HH98] the functions $and_P(X) \mathrel{\widehat{=}} P \wedge X$ and $imp_P(X) \mathrel{\widehat{=}} (P \Rightarrow Y)$ are provided as examples of a Galois connection $(and_P, impl_P)$, that is, $(P \wedge X) \sqsubseteq Y$ **iff** $X \sqsubseteq (P \Rightarrow Y)$.

3.8 Closedness

Another important aspect of predicates in a theory is that they are expected to be combined to allow describe complex examples in that theory. More important, however, is that these combinations of predicates must lie in the original set of valid predicates; operations must be closed with respect to valid predicates. As pointed out by Hoare and He, in program languages these restrictions are usually part of a type theory; for example, if c_1 is a command, c_2 is a command then c_1; c_2 must be command. Another example in the UTP: if we have two designs, their sequential composition must be a design as well. This is called *closedness*.

If we are working with a given theory and we are requested to use, for example, disjunction, it cannot generate a predicate outside the theory set of valid predicates. We are posed to prove that this is true if we expect to use disjunction in our formulas. This is the motivation to show that if some predicates satisfy a given HC the results of operations between these predicates must also satisfy the original HC. In the next chapter, after defining our HCs we show that disjunction, conjunction, conditional, sequential composition and recursion are closed under these HCs, otherwise we could not use them to define richer predicates for object-orientation.

3.9 Conclusions

This chapter presented a brief introduction to the UTP formalism which we use in the following chapters to define our theory for object-orientation, with copy and reference semantics.

In this thesis we present a stepwise integration between four theories: designs (terminating programs), higher order procedures (variables which record procedures), object-orientation (concepts) and pointers (sharing). The big picture of theories integration in this thesis can be resumed by the Figure 3.6.

The starting point is the definition of a theory for object-orientation using designs and higher order procedures theories. Using the pre- and postconditions ($P \vdash Q$) and variables capable of recording procedures we define the concepts of the new theory (subtype, for example) and observational variables (*cls*, *sc*, and *atts*) are introduced to record object-oriented features information.

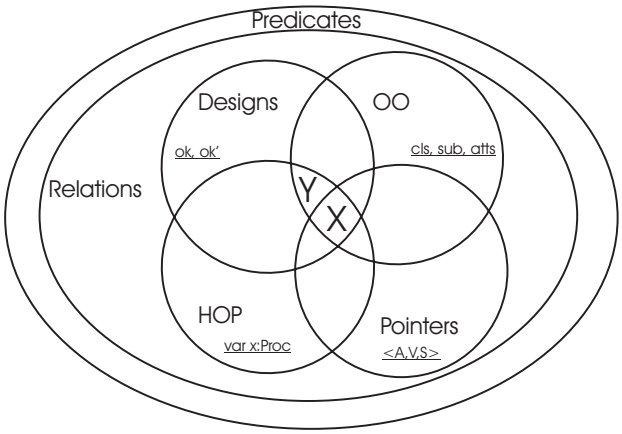

Figure 3.6: Theories integration.

Next we characterise the set of valid values for such variables using healthiness conditions, selecting a subset of relations. The union of regions Y and X represents the set object-oriented programs (or specifications) that are designs, includes higher order variables and also respects the restrictions imposed by the object-oriented theory.

Chapter 4 shows how it is done, and Chapter 5 shows how to merge it with the theory of pointers firstly presented in [CHW06] and extended in [HCW07], thus reducing the set of predicates of interest to X, a theory for object-orientation with pointers.

Chapter 4

Object-Orientation in the UTP

This chapter is organized as follows. In the next section we provide a brief introduction of our approach. Next, in Section 4.2 we introduce the alphabet of our theory, more specifically, the observational variables related to subtyping, inheritance, and dynamic binding. Section 4.3 gives the healthiness conditions and some laws. In Section 4.4, we define class, attribute and method declaration. In Section 4.5, we review the concept of variables, to include type information explicitly. In Section 4.6, we describe well-definedness rules for expressions and the meaning of object creation, type test, type cast, and attribute access. In Section 4.7, we review the semantics of commands emphasizing method call. Finally, in Section 4.8, we provide an overview of what has been achieved so far.

4.1 Introduction

Hoare and He's Unifying Theories of Programming (UTP) has been successfully used to give semantics to several design and programming languages that combine constructs from several paradigms, like, for example, concurrency and time. In this chapter, we study object-oriented concepts present in languages like Java and C++, described in the previous chapters, in the framework of the UTP.

We show how subtyping, data inheritance, (mutually) recursive methods, and dynamic binding can be described in the UTP by combining and extending the theories of designs (terminating programs specified using pre- and postcondition) and higher-order procedures. A distinguishing feature of our approach is modularity: following the style of the UTP, we deal with each concept in isolation; this makes our theory convenient to model integrated languages.

We target general object-oriented concepts, rather than any specific language. We introduce concepts of object-oriented languages progressively and in isolation. We cover subtyping, single inheritance, dynamic binding, and (mutual) recursion. By introducing these features independently,

we provide a general theory of object-orientation that can be combined with other UTP theories in the usual way. At present, we consider copy semantics; reference semantics will be addressed as a next step; as discussed in Chapter 5.

In our theory, a class declaration is not a single block, as usual in object-oriented languages. We have separate constructs to declare a class and its immediate superclass, to declare an attribute, and to declare a method. This follows the approach presented in [Kas05].

Example 2 (Syntax example)**.** By way of illustration, we consider a simple banking system; we define a class *Account*, and its attributes and methods as follows.

$$\begin{array}{l}\textbf{class}\ \mathit{Account};\\ \textbf{att}\ \mathit{Account}\ \mathit{number} : \mathbb{Z}, \mathit{balance} : \mathbb{Z};\\ \textbf{meth}\ \mathit{Account}\ \mathit{credit} = \big(\textbf{val}\ x : \mathbb{Z} \bullet \textbf{self}.\mathit{balance} := \textbf{self}.\mathit{balance} + x\big)\end{array}$$

The declarations of the attributes and methods are independent, and combined in sequence; in particular, the declarations indicate the classes of the attributes and methods that are introduced. This approach simplifies the semantics, and makes the treatment of (mutual) recursion straightforward, as it should be. □

It is well-known that, in the semantics of an object-oriented language, the types of the variables play a central role due to subtyping and dynamic binding [CN00]. In our theory, we have a collection of observational variables that are used to model declarations. They record important typing information and are used to define the semantics of commands. We also drop the UTP assumption that expressions are total; this is not realistic for object-oriented languages due to the possibility of attempts to access attributes and methods of a "**null** object" (that is, "**null** pointer exceptions"). As a consequence, we have to characterize well-defined expressions, and extend the semantics of assignments and conditionals.

Method names are also part of the alphabet of our theory. Their values are parametrised programs [Bac87]. Their treatment follows the approach originally proposed in [Nau95], and adopted in [CN00] to handle methods. It is also the approach followed in the UTP for higher-order procedures.

Dynamic binding is reflected in the value of a method variable: it is a conditional that checks the type of the target object (**self**) and determines the right program that defines the behaviour of the method in each case. In this way, we capture dynamic binding in isolation. This follows the style adopted in an algebraic semantics for object-orientation presented in [BSCC04].

In [SCS06] we presented a preliminary theory of object-orientation in the UTP. Here, we have also presented a brief introduction to the UTP, and revised the semantics to simplify constructs. Moreover, in this thesis we introduce the healthiness conditions that characterise our theory, and prove some of their basic properties as well as closedness of the programming constructs. In the

next chapter, we also present some laws inspired by [BS00, CN00, BSCC04, Cor04] and prove their soundness. Our theory allows us to provide simple and elegant proofs mainly concerning with properties of method declaration and call. Due to our treatment of dynamic binding our proofs follow an algebraic style, as we show in Chapter 6.

4.1.1 Assumptions

Object-oriented features such as attribute overriding, variable shading, and the use of **super** or related notations (to refer to elements of a superclass) are not considered here. They are only syntactic abbreviations that can be easily eliminated by preprocessing.

We consider that the names of classes, attributes, methods (except for method overriding), local variables and parameters are different. This allows us to write simpler predicates while not imposing any relevant practical limitation, both in the semantics and in the laws.

As we have said in the introductory chapters, we are interested in general object-oriented features. By now we postponed the treatment of exceptions, garbage collection or concurrency, for example. These features can be a target of future work.

4.2 Observational Variables

In addition to the programming variables and their dashed counterparts, and to ok and ok' from the theory of designs, our theory includes several extra observational variables. We introduce a variable cls to record the class names; a variable sc to record the subclass relation; and a variable $atts$ to record the names and types of the attributes of every class.

Definition 2 (Classes)**.** The set of classes is recorded in $cls : \mathbb{P}\ name$.

This observational variable allows us to introduce new types other than the primitive ones. For this variable, a special value, useful in the context of examples, is given below.

$$cls^0 = \{\mathbf{Object}\}$$

We consider that ***Object*** is a valid class name.

Definition 3 (Subclasses)**.** The subclass relation is recorded in $sc : name \mapsto name$.

This is a mapping that associates a class name to the name of its immediate superclass. One special value for this variable is defined below.

$$sc^0 = \{\}$$

Usually, in object-oriented languages, ***Object*** does not have a parent. Therefore, it cannot be present in the domain of sc, as constrained by **OO2** presented in the next section.

Using sc, we can define the subtyping relation $A \preceq B$, which holds if A is associated to B in the reflexive and transitive closure sc^* of sc, or if both types are equal (it includes primitives). The inclusion of primitive types allows us to simplify definitions.

$$A \preceq B \mathrel{\widehat{=}} (A, B) \in sc^* \lor A = B$$

The subtyping relation is important in an object-oriented context to establish the well-definedness of assignments and attribute accesses, as we explain in Sections 4.6.1 and 4.7.1. The inclusion of extra primitive types, apart from $\mathbb{B}$ (booleans) and $\mathbb{Z}$ (integers) presented in this work, is not a problem. The strict subtyping relation is denoted by $\prec$, and is defined by

$$A \prec B \mathrel{\widehat{=}} (A, B) \in sc^+$$

Definition 4 (Attributes)**.** The attributes information is recorded in $atts : name \mapsto (name \mapsto Type)$.

This is a mapping from a class name to a description of its attributes that maps each attribute name to its type. Here *Type* stands for any primitive type, or any name in *cls*, that is,

$$Type := \{\mathbb{B}, \mathbb{Z}\} \cup cls$$

Once again, because of ***Object*** special characteristics, we define a special value for this mapping to be used in examples. In this case, we have a mapping which says that the set of attributes for ***Object*** is empty.

$$atts^0 = \{\, \textbf{\textit{Object}} \mapsto \emptyset \}$$

Notice, however, that using the normalization strategy presented in [BSCC04], this set of attributes associated to ***Object*** can be extended.

Definition 5 (Methods)**.** Method names are part of the alphabet of the theory. Their values are texts of parametrised programs (`pds • p`), where `pds` is a list of parameter declarations, and `p` is a program: the body of the parametrised program, which uses the parameters.

We write (`pds • p`) rather than ($pds \bullet p$) to indicate that the values are texts (`pds • p`) of parametrised programs, rather than their meanings ($pds \bullet p$). Value (**val**), result (**res**), and value-result (**valres**) parameters are allowed. The notation `pds` stands for any parameter declaration list, possibly including the three parameter passing mechanisms. For example, **val** $x : X$; **res** $y : Y$; **valres** $z : Z$ is a valid instance of `pds`, where x, y, and z are variable names and X, Y, and Z are their types. The function *types* applied to a list of parameter declarations yields the parameter types as a set. For instance, *types* applied to the example above yields $\{X, Y, Z\}$.

The values of the observational variables named after methods are parametrised nested conditionals with each branch representing the meaning of a method redefinition. For instance, considering that C is a subclass of B, which is itself a subclass of A, and that m is a parameterless method defined in A (with body ma), and redefined in both B and C, with bodies mb and mc, the value of m is

$$\texttt{valres self:Object} \bullet \texttt{mc} \lhd \texttt{self is C} \rhd (\texttt{mb} \lhd \texttt{self is B} \rhd (\texttt{ma} \lhd \texttt{self is A} \rhd \bot_{oo}))$$

Based on the type of the current object (**self**), the nested conditional allows selection of the most specialized version of m. When m is not defined for a given class, then the behaviour of a call to m with an object of this class as a target is unpredictable (the bottom predicate of our theory $\bot_{oo}$, defined in the next section). The type test **self is** N, for a class name N, checks whether the value of **self** is an object of class N, or one of its subclasses. This is why, in the type tests for **self**, the more specialized classes are considered first. In the UTP, programs (and specifications) are predicates; there is no notation to distinguish the text of a program from its semantics. Here, just like in [HH98, Chapter 10] we introduce the distinction. The values of method observational variables have to be texts to allow the use of a syntactic function to capture dynamic binding (see Section 4.4.3).

Finally, for each programming variable x, besides x itself and x', we include in the alphabet two more observational variables xt and xt' to record the declared type of x. This is potentially different from the actual (runtime) type of the value of x, which can be an object of a subclass of the type recorded in xt, when this is a class.

4.3 Healthiness Conditions

We have identified some healthiness conditions of our theory, that is, some predicates that are expected to be valid for all object-oriented specifications (and programs). The first one says that ***Object*** is a valid type.

OO1 $P = P \land \mathit{Object} \in \mathit{cls}$

Our theory relies on a superclass of all classes, represented by the ***Object*** type. An example of a cls instance that satisfy this predicate is cls^0. Also, every class has a parent, except ***Object***.

OO2 $P = P \land \operatorname{dom} sc = \mathit{cls} \setminus \{\mathit{Object}\}$

The top most superclass for all classes is ***Object***, therefore cyclic references are not allowed. Moreover, according to **OO2**, a parent class in sc is necessarily present in cls.

OO3 $P = P \land \forall\, C : \operatorname{dom} sc \bullet (C, \mathit{Object}) \in sc^+$

For all classes present in *cls*, there is a corresponding mapping in *atts* that records the attribute names and types.

OO4 $P = P \land \operatorname{dom} atts = cls$

The names of attributes are different for all classes.

OO5 $P = P \land \forall\, C_1, C_2 : \operatorname{dom} atts \bullet C_1 \neq C_2 \land \operatorname{dom}(atts(C_1)) \cap \operatorname{dom}(atts(C_2)) = \emptyset$

All attributes must have valid types, primitives or defined in *cls*.

OO6 $P = P \land \operatorname{ran}(\bigcup \operatorname{ran} atts) \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup cls$

We name the composition of these conditions $\mathbf{OOI} \mathrel{\widehat{=}} \mathbf{OO1} \circ \mathbf{OO2} \circ \mathbf{OO3} \circ \mathbf{OO4} \circ \mathbf{OO5} \circ \mathbf{OO6}$. These healthiness conditions constrain the initial values of the variables. Predicates in our theory must preserve these properties for the final values of these observational variables; thus we have a similar set of healthiness conditions for the output variables.

OO7 $P = P \land \boldsymbol{Object} \in cls'$

OO8 $P = P \land \operatorname{dom} sc' = cls' \setminus \{\boldsymbol{Object}\}$

OO9 $P = P \land \forall\, C : \operatorname{dom} sc' \bullet (C, \boldsymbol{Object}) \in sc'^{+}$

OO10 $P = P \land \operatorname{dom} atts' = cls'$

OO11 $P = P \land \forall\, C_1, C_2 : \operatorname{dom} atts' \bullet C_1 \neq C_2 \land \operatorname{dom}(atts'(C_1)) \cap \operatorname{dom}(atts'(C_2)) = \emptyset$

OO12 $P = P \land \operatorname{ran}(\bigcup \operatorname{ran} atts') \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup cls'$

The healthiness conditions **OO7-12**, however, can be replaced by the following condition expressed in terms of the identity of our theory. In Law <**OO7-12**,**OO13**-*equivalence*> we prove this equivalence.

OO13 $P = P;\ \mathit{II}_{oo}$

The set of predicates that satisfy all these healthiness conditions is our theory of object-orientation. It is a complete lattice, and its bottom is

$\bot_{oo} \mathrel{\widehat{=}} \mathbf{OO}(\bot)$

where **OO** is the functional composition of **OO1-12**. The identity of our theory, denoted by $I\!I_{oo}$, is the result of the application of **OOI** to the relational identity, $I\!I\ (cls' = cls \wedge sc' = sc \wedge atts' = \dots)$.

$$I\!I_{oo} \mathrel{\widehat{=}} \mathbf{OOI}(I\!I)$$

Our healthiness conditions are idempotent, and the UTP constructs are closed under these conditions. Below, we present some laws that are valid confirming this result; those rules left out are similar. All laws and their proofs are fully presented in the Appendix B.

Law <**OO1**-*idempotent*>
$\mathbf{OO1} \circ \mathbf{OO1} = \mathbf{OO1}$

□

Law <**OO1**-$\wedge$-*closure*>
$\mathbf{OO1}(P \wedge Q) = P \wedge Q$, *provided* P *and* Q *are* **OO1** *healthy.*

□

Law <**OO1**-$\vee$-*closure*>
$\mathbf{OO1}(P \vee Q) = P \vee Q$, *provided* P *and* Q *are* **OO1** *healthy.*

□

Law <**OO1**-$\_ \lhd \_ \rhd \_$-*closure*>
$\mathbf{OO1}(P \lhd b \rhd Q) = P \lhd b \rhd Q$, *provided* P *and* Q *are* **OO1** *healthy.*

□

Law <**OO1**-; -*closure*>
$\mathbf{OO1}(P;\ Q) = P;\ Q$, *provided* P *and* Q *are* **OO1** *healthy.*

□

Law <**OO1**-μ-*closure*>
$\mathbf{OO1}(\mu\, X \bullet F(X)) = \mu\, X \bullet F(X)$, *provided* $F(X)$ *is* **OO1** *healthy.*

□

Similar results are proved for the other healthiness conditions in much the same way. The order of application of **OO1** and **OO2** is irrelevant. This result, the forthcoming laws related to commutativity, and similar results for **OO7-12** prove that there is a subset of relations involving *cls*, *sc* and *atts* where all healthiness conditions **OO1-12** are satisfied. As already said, this subset

is our theory of object-orientation, and we have already shown that the application of conjunction, disjunction, sequence and recursion are closed in this subset.

Law <**OO1**-**OO2**-*commutativity*>
$$\mathbf{OO1} \circ \mathbf{OO2} = \mathbf{OO2} \circ \mathbf{OO1}$$
□

Law <**OO1**-**OO3**-*commutativity*>
$$\mathbf{OO1} \circ \mathbf{OO3} = \mathbf{OO3} \circ \mathbf{OO1}$$
□

Law <**OO1**-**OO4**-*commutativity*>
$$\mathbf{OO1} \circ \mathbf{OO4} = \mathbf{OO4} \circ \mathbf{OO1}$$
□

Law <**OO1**-**OO5**-*commutativity*>
$$\mathbf{OO1} \circ \mathbf{OO5} = \mathbf{OO5} \circ \mathbf{OO1}$$
□

Law <**OO1**-**OO6**-*commutativity*>
$$\mathbf{OO1} \circ \mathbf{OO6} = \mathbf{OO6} \circ \mathbf{OO1}$$
□

Law <**OO2**-**OO3**-*commutativity*>
$$\mathbf{OO2} \circ \mathbf{OO3} = \mathbf{OO2} \circ \mathbf{OO3}$$
□

Law <**OO2**-**OO4**-*commutativity*>
$$\mathbf{OO2} \circ \mathbf{OO4} = \mathbf{OO2} \circ \mathbf{OO4}$$
□

The complete list is in Appendix B. The composition of **OO7**-**OO12** can be replaced by the application of **OO13**.

Law <**OO7-12**,**OO13**-*equivalence*>
$$\mathbf{OO13} = \mathbf{OO7} \circ \mathbf{OO8} \circ \mathbf{OO9} \circ \mathbf{OO10} \circ \mathbf{OO11} \circ \mathbf{OO12}$$
□

4.4 Declarations

In this section we provide the meaning for class, attribute, and method declarations, and some examples. The general form of the declarations are shown in Table 4.1.

Construct	Description
class A **extends** B	Introduces a new class A, subclass of B.
att $A\ x : T$	Introduces an attribute x of type T in the class named A.
meth $A\ m = \Big(pds \bullet p\Big)$	Introduces a method m with formal parameters pds and body p in the class named A.

Table 4.1: Object-oriented declarations.

To each design that we define, we apply the composition of the healthiness conditions **OO1** to **OO12**, that is **OO**, to guarantee that the initial values of the variables are valid and the restrictions over cls', sc', and $atts'$ hold.

4.4.1 Classes

As mentioned before, our aim is to add each feature of object-orientation in isolation. In this direction, a class declaration introduces just a new type, with an empty set of attributes and methods.

Definition 6 (Class introduction)**.** The declaration of a class is defined as shown below.

$$\textbf{class}\ A\ \textbf{extends}\ B \mathrel{\widehat{=}} \textbf{OO}\left(\begin{pmatrix} A \notin Type \land \\ B \in cls \end{pmatrix} \vdash \begin{pmatrix} cls' = cls \cup \{A\} \land \\ sc' = sc \cup \{A \mapsto B\} \land \\ atts' = atts \cup \{A \mapsto \emptyset\} \land \\ w' = w \end{pmatrix}\right)$$

$$\textbf{where}\ w = in\alpha(\textbf{class}\ A\ \textbf{extends}\ B) \setminus \{\, cls,\, sc,\, atts\}$$

The design introduces a record of class A in cls and associates it in the relation sc with B as its immediate superclass. Only new names are allowed ($A \notin Type$), and class B needs to have been previously declared ($B \in cls$). An entry for A in $atts$ is added associated with an empty mapping. No other observational variable w is modified. As explained before, in the UTP, $in\alpha$(**class** A **extends** B) is the input alphabet of the program **class** A **extends** B.

The postcondition of the design establishes new final values for the observational variables of our theory; these values satisfy the properties required by the healthiness conditions **OO7-12**. More specifically, we do not remove ***Object*** from cls' (**OO7** satisfied); the domain of sc is extended with a new class (A) in cls associated with B, and the precondition guarantees that B is already

in *cls*' (**OO8** and **OO9** satisfied); the domain of *atts* is extended to include the new class A introduced in *cls* (**OO10** satisfied); and attribute name is new and has a valid type (**OO11** and **OO12** satisfied). For a simple declaration **class** A, we have the obvious meaning.

$$\textbf{class } A = \textbf{class } A \textbf{ extends } \textit{Object}$$

Example 3 (Class declaration)**.** For our simple banking application, we declare the classes *Account*, which depicts an account of a bank, *BAccount*, an extension of *Account* to hold bonus information, *Contact*, to hold traditional contact information, and *EContact*, an extension of *Contact* to hold electronic contact information. The meaning of the sequence of declarations of these classes is the sequence below.

class *Account*;
class *BAccount* **extends** *Account*;
class *Contact*;
class *EContact* **extends** *Contact*

=

$$\begin{array}{l}
\mathbf{OO}\left(\begin{pmatrix} \mathit{Account} \notin \{\mathbb{B},\mathbb{Z}\} \cup \mathit{cls} \wedge \\ \textbf{\textit{Object}} \in \mathit{cls} \end{pmatrix} \vdash \begin{pmatrix} \mathit{cls}' = \mathit{cls} \cup \{\mathit{Account}\} \wedge \\ \mathit{sc}' = \mathit{sc} \cup \{\mathit{Account} \mapsto \textbf{\textit{Object}}\} \wedge \\ \mathit{atts}' = \mathit{atts} \cup \{\mathit{Account} \mapsto \emptyset\} \end{pmatrix}\right); \\
\mathbf{OO}\left(\begin{pmatrix} \mathit{BAccount} \notin \{\mathbb{B},\mathbb{Z}\} \cup \mathit{cls} \wedge \\ \mathit{Account} \in \mathit{cls} \end{pmatrix} \vdash \begin{pmatrix} \mathit{cls}' = \mathit{cls} \cup \{\mathit{BAccount}\} \wedge \\ \mathit{sc}' = \mathit{sc} \cup \{\mathit{BAccount} \mapsto \mathit{Account}\} \wedge \\ \mathit{atts}' = \mathit{atts} \cup \{\mathit{BAccount} \mapsto \emptyset\} \end{pmatrix}\right); \\
\mathbf{OO}\left(\begin{pmatrix} \mathit{Contact} \notin \{\mathbb{B},\mathbb{Z}\} \cup \mathit{cls} \wedge \\ \textbf{\textit{Object}} \in \mathit{cls} \end{pmatrix} \vdash \begin{pmatrix} \mathit{cls}' = \mathit{cls} \cup \{\mathit{Contact}\} \wedge \\ \mathit{sc}' = \mathit{sc} \cup \{\mathit{Contact} \mapsto \textbf{\textit{Object}}\} \wedge \\ \mathit{atts}' = \mathit{atts} \cup \{\mathit{Contact} \mapsto \emptyset\} \end{pmatrix}\right); \\
\mathbf{OO}\left(\begin{pmatrix} \mathit{EContact} \notin \{\mathbb{B},\mathbb{Z}\} \cup \mathit{cls} \wedge \\ \mathit{Contact} \in \mathit{cls} \end{pmatrix} \vdash \begin{pmatrix} \mathit{cls}' = \mathit{cls} \cup \{\mathit{EContact}\} \wedge \\ \mathit{sc}' = \mathit{sc} \cup \{\mathit{EContact} \mapsto \mathit{Contact}\} \wedge \\ \mathit{atts}' = \mathit{atts} \cup \{\mathit{EContact} \mapsto \emptyset\} \end{pmatrix}\right)
\end{array}$$

The meaning of sequence in our theory is the same as that in the UTP. If we consider that *cls*, *sc*, and *atts* are equal to cls^0, sc^0, and atts^0, respectively, the sequence above specifies the following values for cls', sc' and atts'.

$$\mathit{cls}' = \left\{ \begin{array}{l} \textbf{\textit{Object}}, \\ \mathit{Account}, \\ \mathit{BAccount}, \\ \mathit{Contact}, \\ \mathit{EContact} \end{array} \right\},\ \mathit{sc}' = \left\{ \begin{array}{l} \mathit{Account} \mapsto \textbf{\textit{Object}}, \\ \mathit{BAccount} \mapsto \mathit{Account}, \\ \mathit{Contact} \mapsto \textbf{\textit{Object}}, \\ \mathit{EContact} \mapsto \mathit{Contact} \end{array} \right\}, \text{ and } \mathit{atts}' = \left\{ \begin{array}{l} \textbf{\textit{Object}} \mapsto \emptyset, \\ \mathit{Account} \mapsto \emptyset, \\ \mathit{BAccount} \mapsto \emptyset, \\ \mathit{Contact} \mapsto \emptyset, \\ \mathit{EContact} \mapsto \emptyset \end{array} \right\}$$

□

4.4.2 Attributes

We can introduce attributes in *atts* for those classes already in *cls*.

Definition 7 (Attribute introduction)**.** To introduce an attribute x of type T in class A we can use the construct defined below.

$$\mathbf{att}\ A\ x : T \mathrel{\widehat=} \mathbf{OO}\left(\begin{pmatrix} A \in cls \land \\ x \notin \operatorname{dom} \mathcal{C}(atts, cls) \land \\ T \in Type \end{pmatrix} \vdash \begin{pmatrix} atts' = atts \oplus \{A \mapsto (atts(A) \cup \{x \mapsto T\})\} \land \\ w' = w \end{pmatrix}\right)$$

where $w = in\alpha(\mathbf{att}\ A\ x : T) \setminus \{atts\}$
and $\mathcal{C}(amap, cset) = \bigcup\{N : cset \bullet amap\ N\}$,
$amap$ is an attribute mapping, and $cset$ is class set.

If we try to declare an attribute of a class that has not been declared previously, with a name that was already used, or of a type that is not primitive or present in *cls*, the declaration aborts. The set $\mathcal{C}$ is a useful abbreviation for a mapping of all attributes of any class to their corresponding types, calculated from an attribute mapping as defined for *atts*, and a class set as *cls*. Our healthiness conditions **OO7-12** are not a problem; the design does not change the variables *cls* and *sc*, and the domain of *atts* is not changed.

We can declare several attributes simultaneously, with the obvious meaning.

$$\begin{array}{rcl} \mathbf{att}\ A\ x1 : T1, x2 : T2, \ldots & = & \mathbf{att}\ A\ x1 : T1;\ \mathbf{att}\ A\ x2 : T2;\ \ldots \\ \mathbf{att}\ A\ x1 : T1, B\ x2 : T2, \ldots & = & \mathbf{att}\ A\ x1 : T1;\ \mathbf{att}\ B\ x2 : T2;\ \ldots \end{array}$$

Our notation allows interleaving concerning with the order of class, attribute, and method declaration. For example, the sequence below is allowed.

$$\mathbf{class}\ A;\ \mathbf{att}\ A\ x : \mathbb{Z};\ \mathbf{class}\ B\ \mathbf{extends}\ A;\ \mathbf{att}\ A\ y : \mathbb{B};\ \mathbf{att}\ B\ z : A$$

In this case, the attribute y of class A is declared after the declaration of class B. In fact, if we have recursive classes, the required order of the declaration is different from that adopted in languages where classes are blocks. For example, if a class A has an attribute x whose type is a subclass B of A, then the following order of declaration is required.

$$\mathbf{class}\ A;\ \mathbf{class}\ B\ \mathbf{extends}\ A;\ \mathbf{att}\ A\ x : B$$

Transforming the class-based declarations of an object-oriented language into an appropriate sequence of class and attribute declarations is a simple task. For methods, similar considerations apply; mutual recursion, however, is further discussed in Section 4.7.4.

Example 4 (Attribute declaration)**.** This example adds some attributes to the classes of Example 3.

$$\begin{array}{l} \mathbf{att}\ Account\ number : \mathbb{Z}, balance : \mathbb{Z}, contact : Contact; \\ \mathbf{att}\ BAccount\ bonus : \mathbb{Z}; \\ \mathbf{att}\ Contact\ phone : \mathbb{Z}; \\ \mathbf{att}\ EContact\ icq : \mathbb{Z} \end{array}$$

$=$

$$
\begin{array}{l}
\mathbf{OO}\left(\left(\begin{array}{l} Account \in cls \land \\ number \notin \operatorname{dom}\mathcal{C}(atts, cls) \land \\ \mathbb{Z} \in \{\mathbb{B}, \mathbb{Z}\} \cup cls \end{array}\right) \vdash atts' = atts \oplus \{Account \mapsto atts(Account) \cup \{number \mapsto \mathbb{Z}\}\}\right); \\
\mathbf{OO}\left(\left(\begin{array}{l} Account \in cls \land \\ balance \notin \operatorname{dom}\mathcal{C}(atts, cls) \land \\ \mathbb{Z} \in \{\mathbb{B}, \mathbb{Z}\} \cup cls \end{array}\right) \vdash atts' = atts \oplus \{Account \mapsto atts(Account) \cup \{balance \mapsto \mathbb{Z}\}\}\right); \\
\mathbf{OO}\left(\left(\begin{array}{l} Account \in cls \land \\ contact \notin \operatorname{dom}\mathcal{C}(atts, cls) \land \\ Contact \in \{\mathbb{B}, \mathbb{Z}\} \cup cls \end{array}\right) \vdash atts' = atts \oplus \{Account \mapsto atts(Account) \cup \{contact \mapsto Contact\}\}\right); \\
\mathbf{OO}\left(\left(\begin{array}{l} BAccount \in cls \land \\ bonus \notin \operatorname{dom}\mathcal{C}(atts, cls) \land \\ \mathbb{Z} \in \{\mathbb{B}, \mathbb{Z}\} \cup cls \end{array}\right) \vdash atts' = atts \oplus \{BAccount \mapsto atts(BAccount) \cup \{bonus \mapsto \mathbb{Z}\}\}\right); \\
\mathbf{OO}\left(\left(\begin{array}{l} Contact \in cls \land \\ phone \notin \operatorname{dom}\mathcal{C}(atts, cls) \land \\ \mathbb{Z} \in \{\mathbb{B}, \mathbb{Z}\} \cup cls \end{array}\right) \vdash atts' = atts \oplus \{Contact \mapsto atts(Contact) \cup \{phone \mapsto \mathbb{Z}\}\}\right); \\
\mathbf{OO}\left(\left(\begin{array}{l} EContact \in cls \land \\ icq \notin \operatorname{dom}\mathcal{C}(atts, cls) \land \\ \mathbb{Z} \in \{\mathbb{B}, \mathbb{Z}\} \cup cls \end{array}\right) \vdash atts' = atts \oplus \{EContact \mapsto atts(EContact) \cup \{icq \mapsto \mathbb{Z}\}\}\right)
\end{array}
$$

We use the definition of attribute declaration for each element of the sequence, starting with the attribute *number*, and ending with *icq*. If we suppose that the declaration above comes after the class declarations of Example 3, the expected final value of *atts* is as follows.

$$
atts' = \left\{ \begin{array}{l} \textbf{\textit{Object}} \mapsto \emptyset, \\ Account \mapsto \{number \mapsto \mathbb{Z}, balance \mapsto \mathbb{Z}, contact \mapsto Contact\}, \\ BAccount \mapsto \{bonus \mapsto \mathbb{Z}\}, \\ Contact \mapsto \{phone \mapsto \mathbb{Z}\}, \\ EContact \mapsto \{icq \mapsto \mathbb{Z}\} \end{array} \right\}
$$

□

For a given class N, we define $\mathcal{U}(amap, smap, N)$ to be a mapping that records all the attributes of N, including those declared in its superclasses, considering an attribute mapping *amap*, and a subclass relation *smap* defined with the same types of *atts*, and *sc*, respectively. We define $\mathcal{U}(amap, smap, N)$ as:

$$
\mathcal{U}(amap, smap, N) = \bigcup amap(\!| \ smap^*(\!| \ \{N\} \ |\!) \ |\!)
$$

In words, $\mathcal{U}(atts, sc, N)$ contains all the attribute declarations of all classes related to N by the reflexive and transitive closure of the superclass relation, considering the current attributes in *atts* and subclass relation *sc*. This function is useful to define object creation and also to check if an instance of an object is well-defined.

4.4.3 Methods

For a method declaration to succeed, the class to which it is associated must have been introduced before, and all formal parameters, passed by value (**val**), result (**res**) or value-result (**valres**), must have primitive types or those introduced in *cls*. The result depends on whether the method is being declared for the first time or not. If it is ($m \notin \alpha(\textbf{meth}\ A\ m\ = pds \bullet p)$), then the definition below applies. The new name m is introduced in the alphabet using a variable declaration.

Definition 8 (New method introduction)**.** For new methods, the declaration is defined as follows.

$$\textbf{meth}\ A\ m = \big(pds \bullet p\big) \mathrel{\widehat{=}} \textbf{OO}\left(\textbf{var}\ m;\ \begin{pmatrix} A \in cls \wedge \\ \forall\, t \in types(pds) \bullet t \in Type \end{pmatrix} \vdash \begin{pmatrix} m' = \mathtt{pds}_e \bullet (\mathtt{p} \lhd \mathtt{self\ is\ A} \rhd \bot_{oo}) \wedge \\ w' = w \end{pmatrix}\right)$$

provided $m \notin \alpha(\textbf{meth}\ A\ m = \big(pds \bullet p\big))$
where $\mathtt{pds}_e = \mathtt{valres\ self{:}Object;\ pds}$ **and** $w = in\alpha(\textbf{meth}\ A\ m = \big(pds \bullet p\big))$

The value of m' is the text of a parametrised program. Methods are higher-order, parametrised program-valued variables, much in the same way as in the theory of higher-order procedures and parameters of the UTP. The parameters of m' are those in *pds* and an extra parameter **self** to represent the target of a call; its type is ***Object***. We use the notation $\mathtt{pds}_e$ to represent this extended parameter list. Just as with **var** x, which introduces in the alphabet new variables x and x', for **meth** $A\ m$, we introduce in the alphabet the variables m and m', and use a design to define the value of m'.

For the case of a redefinition of a method m ($m \in \alpha(\textbf{meth}\ A\ m = pds \bullet p)$) we have another definition.

Definition 9 (Method redefinition)**.** If the method name declared is not new, the corresponding definition is the following.

$$\textbf{meth}\ A\ m = \big(pds \bullet p\big) \mathrel{\widehat{=}} \textbf{OO}\left(\begin{pmatrix} A \in cls \wedge \\ \exists\, \mathtt{q} \bullet m = \mathtt{pds}_e \bullet \mathtt{q} \end{pmatrix} \vdash \left(\exists\, \mathtt{q} \bullet \begin{pmatrix} m = \mathtt{pds}_e \bullet \mathtt{q} \wedge \\ m' = \mathtt{pds}_e \bullet join(A, \mathtt{p}, \mathtt{q}) \wedge \\ w' = w \end{pmatrix}\right)\right)$$

provided $m \in \alpha(\textbf{meth}\ A\ m = \big(pds \bullet p\big))$
where $\mathtt{pds}_e = \mathtt{valres\ self{:}Object;\ pds}$, $w = in\alpha(\textbf{meth}\ A\ m = \big(pds \bullet p\big)) \setminus \{m\}$
and

$join(A, \mathtt{p}, \bot_{oo}) = \mathtt{p} \lhd \mathtt{self\ is\ A} \rhd \bot_{oo}$

$$join(A, \mathtt{p}, \mathtt{q}_l \lhd \mathtt{self\ is\ B} \rhd \mathtt{q}_r) = \begin{cases} \mathtt{p} \lhd \mathtt{self\ is\ A} \rhd (\mathtt{q}_l \lhd \mathtt{self\ is\ B} \rhd \mathtt{q}_r), & \text{if } A \prec B \\ \mathtt{q}_l \lhd \mathtt{self\ is\ B} \rhd join(A, \mathtt{p}, \mathtt{q}_r) & \text{, otherwise} \end{cases}$$

$join(A, \mathtt{p}, \mathtt{q}) = \bot_{oo}$, *for programs* $\mathtt{q}$ *of every other form*

If the method declaration is a redefinition, the method signature must be exactly the same as that of the existing method, and a new conditional is built to take into account the class hierarchy. The definition of the syntactic function *join* deals with redefinition of m both in superclasses and in subclasses of the class where the original definition is placed. The use of *join* allows us to introduce the method values as (parametrised) programs in a form where dynamic binding is already resolved, as in algebraic methods [BS00, BSCC04] and in the weakest precondition approach [CN00]. As already said, the special variable **self** denotes the target of the method call. All references to attributes in method bodies must be prefixed with **self**; variables without this prefix are formal parameters or local variables.

We give the meaning of a parametrised program as a function from a value or a variable name to a program (or predicate). We consider each of the mechanisms of parameter passing individually; the definitions reflect the standard way of implementing them.

For a value parameter, the semantics is a higher-order function that takes the value of the argument and gives the program that declares the formal parameter as a local variable and initializes it with the argument.

$$(\mathbf{val}\ v : T \bullet p) \mathrel{\widehat{=}} (\lambda\, w : T \bullet (\mathbf{var}\ v : T;\ v := w;\ p;\ \mathbf{end}\ v : T))$$

A function that models a parametrised program with a parameter passed by result takes as argument the name of a variable: an element of the syntactic category $\mathcal{N}$. This is the argument in a method call.

$$(\mathbf{res}\ v : T \bullet p) \mathrel{\widehat{=}} (\lambda\, w : \mathcal{N} \bullet (\mathbf{var}\ v : T;\ p;\ w := v;\ \mathbf{end}\ v : T))$$

In this case, the local variable corresponding to the formal parameter is not initialized; its value is assigned to the argument.

For a value-result parameter, the definition is as expected: the local variable is initialized and then assigned to the argument in the end.

$$(\mathbf{valres}\ v : T \bullet p) \mathrel{\widehat{=}} (\lambda\, w : \mathcal{N} \bullet (\mathbf{var}\ v : T;\ v := w;\ p;\ w := v;\ \mathbf{end}\ v : T))$$

The parameter of the function is again a program variable.

Lambda-reduction is extended to cope with variable parameters: elements of the syntactic category $\mathcal{N}$. It is an abstraction over four arguments: a variable, the corresponding type variable, and their dashed counterparts. A similar semantics for parametrisation was presented in [CSW05].

$$(\lambda\, x : \mathcal{N} \bullet p)(y) \mathrel{\widehat{=}} p\,[y, y', yt, yt'/x, x', xt, xt']$$

Example 5 (Method declaration)**.** In this example we show the semantics of method declarations, considering that *cls* is the one defined in Example 3, and *atts* is that defined in Example 4. We

introduce a method *credit* for *Account*, and we redefine it for class *BAccount* to also increase the value of a bonus variable.

$$\textbf{meth}\ Account\ credit = \big(\textbf{val}\ x : \mathbb{Z} \bullet \textbf{self}.balance := \textbf{self}.balance + x\big)\,;$$
$$\textbf{meth}\ BAccount\ credit = \big(\textbf{val}\ x : \mathbb{Z} \bullet \textbf{self}.bonus := \textbf{self}.bonus + 1;\ \textbf{self}.balance := \textbf{self}.balance + x\big)$$

We observe that, in the body of the redefinition of *credit* for *BAccount*, we have a repetition of the code in the body of *credit* as defined for *Account*. In a programming language, this is likely to be written as **super**.*credit*(x) or using some other similar notation that avoids code repetition. As we explained in Section 4.2, however, semantically, these constructs can be removed using a copy rule. For this reason, we do not consider this issue here. The meaning of the two method declarations is given by the sequence below.

$$\textbf{OO}\left(\textbf{var}\ credit;\ \left(Account \in cls \vdash credit' = \left(\begin{array}{l}\texttt{valres self:Object; val x:}\mathbb{Z} \bullet \\ \quad \texttt{self.balance...} \lhd \texttt{self is Account} \rhd \bot_{oo}\end{array}\right)\right)\right);$$
$$\textbf{OO}\left(\begin{array}{l}\left(\begin{array}{l}BAccount \in cls \wedge \\ \exists\, \texttt{q} \bullet credit = (\texttt{valres self:Object; val x:}\mathbb{Z} \bullet \texttt{q})\end{array}\right) \\ \vdash \\ \left(\exists\, \texttt{q} \bullet \left(\begin{array}{l}credit = (\texttt{valres self:Object; val x:}\mathbb{Z} \bullet \texttt{q}) \wedge \\ credit' = \texttt{valres self:Object; val x:}\mathbb{Z} \bullet join\,\big(BAccount, (\texttt{self.bonus :=...}), \texttt{q}\big)\end{array}\right)\right)\end{array}\right)$$

If we eliminate the sequential composition, the value of the variable `q` existentially quantified in the second design is determined to be the body of *credit* as defined in the first design. With that, we can calculate the result of *join*. The final value of *credit* is of the following form.

```
valres self:Object; val x:ℤ •
   self.bonus := self.bonus + 1;... ◁ self is BAccount▷
      (self.balance := self.balance + x ◁ self is Account ▷ ⊥oo)
```

The conditional generated by *join* selects the appropriate command depending on the type of **self**. This is the expected behaviour in the presence of dynamic binding. □

4.5 Variables

In [HH98], type information is not explicitly recorded for the variables. In an object-oriented language, where types play a central role, this is not appropriate. In our theory, the values of the variables are pairs, whose first element is the (runtime) type of the variable, and the second is the value itself. We define the construct **var** $x : T$ for typed declaration of variables, where T is the static type of the variable x.

Definition 10 (Variable declaration)**.**

$$\textbf{var}\ x : T \mathrel{\widehat=} \textbf{OO}(\{T \in Type\}_\perp;\ \textbf{var}\ xt, x;\ \textbf{true} \vdash xt' = T \wedge x' \in \mathcal{V}(T) \wedge w' = w)$$

$$\textbf{provided}\ x \notin in\alpha(\textbf{var}\ x : T)\ \textbf{and}\ w = in\alpha(\textbf{var}\ x : T)$$

We use the existing **var** construct to introduce both x and xt in the alphabet. In the definition, we require that T is a valid type, if it is invalid the sequential composition aborts. For that, we use an assertion, defined in Chapter 3. Otherwise, the type xt of x is defined to be T, and an arbitrary element of T is chosen as its initial value. All the other variables are not changed. In assignments to x its value, which is a pair (e_t, e_v), may change, but xt does not.

To complete this definition, we need to define the set of elements $\mathcal{V}(C)$ of a type C. These are pairs in which the first element is a subclass A of C, possibly C itself, and the second element is either the special value **null** or a mapping (record) that associates a value to the name of each of the attributes of A, in the case of classes. For primitives types the second value is a primitive value, such as 1, for integer, or **true** for booleans. A formal definition is a function that takes sc and $atts$ as parameters; a similar function is specified in [CN00]. As with **var** x, our typed declaration is a non-homogeneous relation: the alphabet of **var** $x : T$ does not include x or xt.

The definition of **end** $x : T$ (the construct used to finalize the scope of x) is similar to that in the UTP for **end** x. There are no concerns with type at the end of the scope of a variable, but we need to close the scope of both x and xt.

Definition 11 (Variable removal)**.**

$$\textbf{end}\ x : T \mathrel{\widehat{=}} \mathbf{OO}(\textbf{end}\ x, xt)$$

The discussion about the structure of values is extremely important to the definition of object value, and the correctness of assignments and method calls. This interpretation of variables and values is not against the principles of the UTP; we have just made explicit the representation of values in order to handle the concepts of object-orientation.

4.6 Expressions

In this section, we specify well-definedness rules for expressions, and the semantics of object creation, type test, type cast, and attribute accesses.

4.6.1 Well-definedness

Our theory includes new forms of expression e characterised by the BNF-like definition in Table D.1. There, v is a primitive or object value. The expressions le, named left expressions are

$$\begin{array}{rcl} e & ::= & v \mid le \mid \textbf{new}\ N \mid e\ \textbf{is}\ N \mid (N)e \mid f(e) \mid \textbf{null} \\ le & ::= & x \mid \textbf{self} \mid le.x \end{array}$$

Table 4.2: BNF for object-oriented expressions.

ordinary variable names or the special variable named **self**, followed by a (possibly empty) sequence of dot-separated names. The expression **new** N is an object creation, e **is** N is a type test, and $(N)e$ is a type cast. There is also a group of built-in operations over expressions, like, for instance, arithmetic and relational operators, denoted by $f(e)$.

For an expression e, we write e_t to denote the first element of the value of e, and e_v to denote the second element. In other words, e_t is the type of the value of e, and e_v is the value itself forming a pair (e_t, e_v). The construct **null** actually stands for a family of values, one for each class. The type held by e_t in this case is inferred from the context. For instance, in an assignment $x := \textbf{null}$, we have that $e_t = xt$; this means that the runtime type of **null** is the declared type of x.

The well-definedness of expressions, and commands, is specified by a function named $\mathcal{D}$. If an expression has a primitive value, it is well-defined if the value belong to the set of possible values of the type. For objects, we must check if the type belongs to *cls*, and if the value belongs to the set of values of type T, $\mathcal{V}(T)$. For primitive types the test is simpler.

Primitive Values			**Objects**		
$\mathcal{D}((\mathbb{B}, v))$	$\widehat{=}$	$v \in \mathbb{B}$	$\mathcal{D}((T, \textbf{null}))$	$\widehat{=}$	$T \in cls$
$\mathcal{D}((\mathbb{Z}, v))$	$\widehat{=}$	$v \in \mathbb{Z}$	$\mathcal{D}((T, v))$	$\widehat{=}$	$T \in cls \land (T, v) \in \mathcal{V}(T)$

Variables are well-defined if their types are either primitive or present in *cls*. If a variable has the special name **self**, it cannot be of a primitive type.

Variables

$$\begin{array}{rcl} \mathcal{D}(x) & \widehat{=} & xt \in Type \\ \mathcal{D}(\textbf{self}) & \widehat{=} & \textbf{self}t \in cls \end{array}$$

An attribute access $le.x$ is valid only if le is well-defined, the type of le is not primitive, the value of le is different from **null**, and x is in the domain of the value of le.

Attribute Accesses

$$\mathcal{D}(le.x) \;\widehat{=}\; \mathcal{D}(le) \land le_t \in cls \land le_v \neq \textbf{null} \land x \in \operatorname{dom} le_v$$

A **new** N declaration is valid only if the class N is recorded in *cls*. A type test e **is** N or casting (N) e can be done only if e is a well-defined expression and N is not primitive. For a type cast, the expression has to be of a valid subtype of N.

Typing

$$\begin{array}{rcl} \mathcal{D}(\textbf{new } N) & \widehat{=} & N \in cls \\ \mathcal{D}(e \textbf{ is } N) & \widehat{=} & \mathcal{D}(e) \land N \in cls \\ \mathcal{D}((N)e) & \widehat{=} & \mathcal{D}(e) \land N \in cls \land e_t \preceq N \end{array}$$

The well-definedness restrictions for built-in operations for primitive types, $f(e)$, are defined individually and are very similar. We show the example of the remainder of a division operator,

usually written '%' in programming languages.

Remainder
$$\mathcal{D}(x \% y) \quad \widehat{=} \quad \mathcal{D}(x) \land \mathcal{D}(y) \land x_t = \mathbb{Z} \land y_t = \mathbb{Z} \land y_v \neq 0$$

In Section 4.7.1, we use the function $\mathcal{D}$ on expressions to define well-definedness rules for commands.

4.6.2 Object Creation

An object value is a pair $(type, value)$: the *type* is a class name and the *value* is a mapping from names to attribute values. Using *sc* and *atts* to recover attributes and inheritance information, we provide a definition for **new** as follows.

$$\textbf{new } N \mathrel{\widehat{=}} \left(N, \left\{ \begin{pmatrix} x : \operatorname{dom} map; \\ t : Type; \\ v : \{\, T : Type;\ i : T \bullet i \,\} \end{pmatrix} \mid \left(\begin{pmatrix} map(x) = \mathbb{B} \land \\ t = \mathbb{B} \land \\ v = \textbf{false} \end{pmatrix} \lor \begin{pmatrix} map(x) = \mathbb{Z} \land \\ t = \mathbb{Z} \land \\ v = 0 \end{pmatrix} \lor \left(\exists\, T : cls \bullet \begin{pmatrix} map(x) = T \land \\ t = T \land \\ v = \textbf{null} \end{pmatrix} \right) \right) \bullet x \mapsto (t, v) \right\} \right)$$

where $map = \mathcal{U}(atts, sc, N)$

This definition says that the value of a newly created object is a mapping from attribute names x to values (t, v) that associate all boolean attributes to **false**, all integer attributes to 0, and all class-typed attributes to **null**. For example, the value of **new** *BAccount*, considering the values of *sc* and *atts* obtained after the declarations of Examples 3 and 4, is

$$(\ BAccount, \{number \mapsto (\mathbb{Z}, 0), balance \mapsto (\mathbb{Z}, 0), contact \mapsto (Contact, \textbf{null}), bonus \mapsto (\mathbb{Z}, 0)\}\ )$$

In this example, all attributes from class *Account* (*number*,*balance*,*contact*), as well as those from *BAccount* (*bonus*), are included.

4.6.3 Type Test

The expression e **is** N is a boolean that indicates whether the value of e belongs to the class N or to one of its subclasses. The result yielded by such an expression is

$$e \textbf{ is } N \mathrel{\widehat{=}} (\mathbb{B}, e_t \preceq N)$$

For example, (**new** *BAccount*) **is** *Account*
$$\begin{aligned} &= (BAccount, \{\dots\}) \textbf{ is } Account \\ &= (\mathbb{B}, BAccount \preceq Account) \\ &= (\mathbb{B}, \textbf{true}) \end{aligned}$$

This is justified by the definitions of **new**, type test, and $\preceq$, if we assume that *cls* and *sc* are as defined in Example 3.

4.6.4 Type Cast

The result of a cast $(N)e$ is the expression e itself, if the casting is well defined. Since we are only defining the meaning of well-defined expressions, our specification is surprisingly trivial.

$$(N)e \;\widehat{=}\; e$$

For example, provided that $BAccount \preceq Account$

$$\begin{aligned} (Account)\ \textbf{new}\ BAccount &= (Account)(BAccount, \{\dots\}) \\ &= (BAccount, \{\dots\}) \end{aligned}$$

In the semantics of assignments and conditionals, well-definedness is checked.

4.6.5 Attribute Access

An attribute access $le.x$ recovers from the object value mapping (le_v) the attribute named x.

$$le.x \;\widehat{=}\; le_v(x)$$

Again, we have a very simple definition, because we are only considering well-defined attribute accesses. For example, suppose that we have an instance of *BAccount* such as in the following program.

var $x : BAccount$;
$x :=$ **new** $BAccount$;

The result of the expression $x.bonus$ is given by

$$\begin{aligned} x.bonus &= (x_t, x_v).bonus \\ &= (BAccount, \{\dots, bonus \mapsto (\mathbb{Z}, 0)\}).bonus \\ &= \{\dots, bonus \mapsto (\mathbb{Z}, 0)\}(bonus) \\ &= (\mathbb{Z}, 0) \end{aligned}$$

As expected, we select the value associated to *bonus* in the mapping of attribute values for x. If we have a composite name like $le.x.y$, we successively apply the lookup – $(le_v(x)).y$ – to select the expected value.

4.7 Commands

Our theory includes assignments $le := e$ of a value e to a left expression le, and method calls $le.m(a)$ with target le and list of arguments a. Moreover, since expressions have changed, we need to consider well-definedness for some commands. We also introduce mutual recursion. Sequence remains unchanged.

$$c ::= le := e \mid \mathit{II} \mid \textbf{var}\ x : T \mid \textbf{end}\ x : T \mid c_1 \lhd e \rhd c_2 \mid c_1;\ c_2 \mid \mu\, X \bullet F(X) \mid le.m(e)$$

Table 4.3: BNF for object-oriented commands.

4.7.1 Well-definedness

In this section, we specify well-definedness for assignments, conditionals, and method calls. We consider two forms of assignment: assignments to variables, and assignments to object attributes. An assignment of an expression e to a variable x is considered well-defined if x is well-defined, e is well-defined, and the type of e is a subtype of the type xt of x.

Assignment to variables
$\mathcal{D}(x := e) \ \widehat{=}\ \mathcal{D}(x) \wedge \mathcal{D}(e) \wedge e_t \preceq xt$

For an assignment of an expression e to an attribute x of le to be well-defined, the expression $le.x$ must be well-defined, e must be well-defined, and the type of e must be a subtype of $\mathcal{U}(atts, sc, le_t)(x)$, the type of the attribute x in the class le_t.

Assignment to attributes
$\mathcal{D}(le.x := e) \ \widehat{=}\ \mathcal{D}(le.x) \wedge \mathcal{D}(e) \wedge e_t \preceq \mathcal{U}(atts, sc, le_t)(x)$

For a conditional to be well-defined, the condition must be well-defined and yield a boolean value.

Conditional
$\mathcal{D}(P \lhd e \rhd Q) \ \widehat{=}\ \mathcal{D}(e) \wedge e_t = \mathbb{B}$

The definition of well-definedness for method calls is the most extensive. A method call of the form $le.m(a)$ is valid if:

- le is well-defined;
- the value of le is different from **null**;
- the method m is defined for the type of le;
- to avoid aliasing, le is not passed as an argument and is not involved in any argument. For further details about this restriction, see [CN00];

- the types of the arguments in the list a must be compatible with the formal parameters list of m.

We present well-definedness definitions according to the parameter passing mechanism. Starting with value parameters, we have the definition below.

Method call
$\mathcal{D}(le.m(e)) \mathrel{\widehat{=}} \mathcal{D}(le) \land le_v \neq \textbf{null} \land compatible(le, m) \land e_t \preceq T$

provided $\exists \mathtt{p} \bullet m = (\mathtt{val\ x{:}T} \bullet \mathtt{p})$,
where $compatible(le, m) \mathrel{\widehat{=}} \exists \mathtt{pds}, \mathtt{p} \bullet m = (\mathtt{pds} \bullet \mathtt{p}) \land le_t \in scan(\mathtt{p})$
with
$scan(\bot_{oo}) = \{\}$
$scan(\mathtt{p}_l \lhd \mathtt{self\ is\ A} \rhd \mathtt{p}_r) = \{B : cls \mid B \preceq A\} \cup scan(\mathtt{p}_r)$

The *scan* function yields the set of class names for which the method m may have a definition different from $\bot_{oo}$. For result and value-result parameters, we use the function *sdisjoint* [CN00], which verifies if *le* is involved in any of the arguments.

$\mathcal{D}(le.m(y)) \mathrel{\widehat{=}} \mathcal{D}(le) \land le_v \neq \textbf{null} \land compatible(le, m) \land sdisjoint(le, y) \land T \preceq y_t$
provided $\exists \mathtt{p} \bullet m = (\mathtt{res\ x{:}T} \bullet \mathtt{p})$

$\mathcal{D}(le.m(z)) \mathrel{\widehat{=}} \mathcal{D}(le) \land le_v \neq \textbf{null} \land compatible(le, m) \land sdisjoint(le, z) \land T = z_t$
provided $\exists \mathtt{p} \bullet m = (\mathtt{valres\ x{:}T} \bullet \mathtt{p})$

The definition for a method call with multiple arguments is a straightforward extension of these definitions.

4.7.2 Assignments

Now we define assignments to variables, and assignments to attributes of object variables. In our theory, we observe that modifying the value of method variables, type variables *xt*, *cls*, *sc*, *atts*, or *ok* is not allowed, in much the same way that assignments to *ok* are not allowed in the theory of designs as well.

If we establish the well-definedness of an assignment, we can update the value of the variable to be that of the expression.

$x := e \mathrel{\widehat{=}} \mathbf{OO}(\mathcal{D}(x := e) \vdash x' = e \land w' = w)$

where $w = in\alpha(x := e) \setminus \{x\}$

For example, given a variable x of type *Account* ($xt = Account$), we can calculate the meaning of the assignment $x :=$ **new***BAccount* as follows, provided that y is the list of undashed variables in the alphabet, other than x, and that *cls*, *sc* and *atts* are as in Examples 3 and 4.

$x :=$ **new** *BAccount*

$$
\begin{array}{l}
=\{ \text{ assignment } \} \\
\mathbf{OO}\,(\ \mathcal{D}(x := (BAccount, \{number \mapsto \dots\})) \ \vdash\ x' = (BAccount, \{number \mapsto \dots\}) \wedge y' = y\ )
\end{array}
$$

$$
\begin{array}{l}
=\{ \mathcal{D} \text{ for variable assignments } \} \\
\mathbf{OO}\left(\left(\begin{array}{l} \mathcal{D}(x) \wedge \\ \mathcal{D}((BAccount, \{number \mapsto \dots\})) \wedge \\ BAccount \preceq Account \end{array}\right) \vdash\ x' = (BAccount, \{number \mapsto \dots\}) \wedge y' = y\right)
\end{array}
$$

$$
\begin{array}{l}
=\{ \mathcal{D} \text{ for variables and instances; assumptions on } cls \text{ and subtyping } \} \\
\mathbf{OO}\left(\left(\begin{array}{l} Account \in \{\mathbb{B}, \mathbb{Z}\} \cup cls \wedge \\ BAccount \in cls \wedge \\ (BAccount, \{number \mapsto \dots\}) \in \mathcal{V}(BAccount) \end{array}\right) \vdash\ x' = (BAccount, \{number \mapsto \dots\}) \wedge y' = y\right)
\end{array}
$$

$$
\begin{array}{l}
=\{ \text{ assumptions on } cls \text{; and definition of } \mathcal{V} \ \} \\
\mathbf{OO}\,(\mathbf{true} \vdash\ x' = (BAccount, \{number \mapsto \dots\}) \wedge y' = y\ )
\end{array}
$$

To update an attribute of an object-valued expression, we check the well-definedness of the assignment, and if it is valid, then we update the mapping that records the attribute value, maintaining the left expression type unchanged.

$$le.x := e \mathrel{\widehat{=}} \mathbf{OO}(\mathcal{D}(le.x := e) \vdash le' = (le_t, le_v \oplus \{x \mapsto e\}) \wedge w' = w)$$

$$\textbf{where } w = in\alpha(le.x := e) \setminus \alpha(le)$$

We use $\alpha(le)$ to denote a variable in the alphabet whose value is being inspected by the left-expression le. If le is a variable, then $\alpha(le)$ is the variable itself. On the other hand, for $x.y$ and $x.y.z$, for example, the result is x. The equality $le' = (le_t, le_v \oplus \{\, x \mapsto e \,\})$ for the case in which le is itself an attribute access $y.z$, for instance, is an abbreviation for the equality $y' = (y_t, y_v \oplus \{\, z \mapsto y.z \oplus \{\, x \mapsto e \,\}\,\})$.

For example, given a variable x of type $Account$ ($xt = Account$), which has been initialized with **new** $BAccount$ ($x = (BAccount, \{number \mapsto (\mathbb{Z}, 0), \dots\})$), we can describe the attribute update $x.number := 1$ as follows, provided that y is the list of undashed variables in the alphabet, other than x, and cls, sc, $atts$ are as in Examples 3 and 4.

$$x.number := 1$$

$$
\begin{array}{l}
=\{ \text{ assignment to attributes } \} \\
\mathbf{OO}\left(\begin{array}{l} \mathcal{D}((BAccount, \{number \mapsto (\mathbb{Z}, 0), \dots\}).number := (\mathbb{Z}, 1)) \\ \qquad \vdash \big(x' = (BAccount, \{number \mapsto (\mathbb{Z}, 0), \dots\} \oplus \{number \mapsto (\mathbb{Z}, 1)\}) \wedge y' = y\big) \end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
=\{ \mathcal{D} \text{ for attribute assignments; and mapping replacement } \} \\
\mathbf{OO}\left(\left(\begin{array}{l} \mathcal{D}((BAccount, \{number \mapsto (\mathbb{Z}, 0), \dots\}).number) \wedge \\ \mathcal{D}((\mathbb{Z}, 1)) \wedge \\ \mathbb{Z} \preceq \mathcal{U}(atts, sc, Account)(number) \end{array}\right) \vdash \left(\begin{array}{l} x' = (BAccount, \{number \mapsto (\mathbb{Z}, 1), \dots\}) \\ \wedge \\ y' = y \end{array}\right)\right)
\end{array}
$$

$$
\begin{array}{l}
=\{\ \mathcal{D} \text{ for attribute accesses and primitive instances; and definition of } \mathcal{U}\ \} \\
\mathbf{OO}\left(\begin{pmatrix}\mathcal{D}((BAccount, \{number \mapsto (\mathbb{Z}, 0), \dots\})) \wedge \\ BAccount \in cls \wedge \\ \{number \mapsto (\mathbb{Z}, 0), \dots\} \neq \mathbf{null} \wedge \\ number \in \mathrm{dom}\{number \mapsto (\mathbb{Z}, 0), \dots\} \wedge \\ \mathbb{Z} \preceq \mathbb{Z}\end{pmatrix} \vdash \begin{pmatrix} x' = (BAccount, \{number \mapsto (\mathbb{Z}, 1), \dots\}) \\ \wedge \\ y' = y\end{pmatrix}\right)
\end{array}
$$

$$
\begin{array}{l}
=\{\ \mathcal{D} \text{ for object instance; set properties; and assumptions on } cls \text{ and subtyping}\ \} \\
\mathbf{OO}\left(\begin{pmatrix}BAccount \in cls \wedge \\ (BAccount, \{number \mapsto \dots\}) \in \mathcal{V}(BAccount)\end{pmatrix} \vdash x' = (BAccount, \{number \mapsto (\mathbb{Z}, 1), \dots\}) \wedge y' = y\right)
\end{array}
$$

$$
\begin{array}{l}
=\{\ \text{definition of } \mathcal{V}\text{; and set properties}\ \} \\
\mathbf{OO}\left(\mathbf{true} \vdash x' = (BAccount, \{number \mapsto (\mathbb{Z}, 1), \dots\}) \wedge y' = y\right)
\end{array}
$$

If we had not initialized the variable x, the assignment would not be well-defined and would abort.

4.7.3 Conditional

We need to redefine the conditional to consider the well-definedness of the condition.

$$P \lhd e \rhd Q \mathrel{\widehat{=}} \mathbf{OO}(\mathcal{D}(P \lhd e \rhd Q) \wedge ((e_v \wedge P) \vee (\neg e_v \wedge Q)))$$

For example, suppose we have the variables *cls*, *sc* and *atts* as in the Examples 3 and 4. If we declare a variable **var self** : *Account*, and initialize it as **self** := **new** *BAccount*, the result of the conditional $P \lhd \mathbf{self\ is}\ BAccount \rhd Q$, with arbitrary P and Q, is P, as shown below.

$$\mathbf{OO}\,(P \lhd \mathbf{self\ is}\ BAccount \rhd Q)$$

$$
\begin{array}{l}
=\{\ \text{definition of conditional; and definition of type test}\ \} \\
\mathbf{OO}\left(\mathcal{D}(P \lhd \mathbf{self\ is}\ BAccount \rhd Q) \wedge \begin{pmatrix}(\mathbb{B}, (BAccount, \{number \mapsto \dots\})_t \preceq BAccount)_v \wedge P) \\ \vee \\ (\neg(\mathbb{B}, (BAccount, \{number \mapsto \dots\})_t \preceq BAccount)_v \wedge Q)\end{pmatrix}\right)
\end{array}
$$

$$
\begin{array}{l}
=\{\ \text{type selection}\ \} \\
\mathbf{OO}\left(\mathcal{D}(P \lhd \mathbf{self\ is}\ BAccount \rhd Q) \wedge \begin{pmatrix}(\mathbb{B}, BAccount \preceq BAccount)_v \wedge P) \\ \vee \\ (\neg(\mathbb{B}, BAccount \preceq BAccount)_v \wedge Q)\end{pmatrix}\right)
\end{array}
$$

$$
\begin{array}{l}
=\{\ \mathcal{D} \text{ for conditionals; value selection; assumptions on } cls \text{ and subtyping}\ \} \\
\mathbf{OO}\left(\mathcal{D}(\mathbf{self\ is}\ BAccount) \wedge (\mathbb{B}, (BAccount, \{number \mapsto \dots\})_t \preceq BAccount)_t = \mathbb{B} \wedge \begin{pmatrix}(\mathbf{true} \wedge P) \\ \vee \\ (\mathbf{false} \wedge Q)\end{pmatrix}\right)
\end{array}
$$

$=\{$ $\mathcal{D}$ for type test; type selection; and $\wedge$-$\vee$-elimination $\}$
$\mathbf{OO}(\mathcal{D}(\mathbf{self}) \wedge \mathit{BAccount} \in \mathit{cls} \wedge \mathbb{B} = \mathbb{B} \wedge P)$

$=\{$ $\mathcal{D}$ for **self**; assumptions on cls $\}$
$\mathbf{OO}(\mathbf{self}t \in \mathit{cls} \wedge P)$

$=\{$ assumptions on cls $\}$
$\mathbf{OO}(P)$

If the type test were **false**, the branch selected would be Q. Moreover, according to the well-definedness rules for the variable **self**, it cannot be an instance of a primitive type. If this were the case, the meaning of the conditional would be abort. It may be the case that P is not well-defined; in this case, abortion arises from the definition of P.

4.7.4 Recursion

Basically, the meaning of recursion is as in the UTP: defined in terms of least fixed points. The complete lattice is that of parametrised programs, with refinement as the partial order. For each parameter declaration, the set of programs with those parameters is a complete lattice; refinement is defined pointwise [BvW98].

Definition 12 (Recursive Method)**.** The general form for the declaration of a recursive method m of class A is the following.

$$\mathbf{meth}\ A\ m = \mu\, X \bullet \left(\ \mathit{pds} \bullet F(X)\ \right)$$

For example, a method to calculate factorials could be added to a class A as follows

$$\mathbf{meth}\ A\ m = \mu\, X \bullet \big(\mathbf{val}\ n : \mathbb{Z};\ \mathbf{res}\ r : \mathbb{Z} \bullet r := 1 \lhd n \leq 0 \rhd r := n * X(n-1, r)\big)$$

This is not in conflict with the expected form of a method declaration $\mathbf{meth}\ A\ m = \Big(\mathit{pds} \bullet p\Big)$, since, of course, the least fixed point operator results in a parametrised program. In particular, the parameters are the same as those in the body of the recursion. For each parameter declaration, we take the fixed point in the lattice of parametrised programs with those parameters.

Definition 13 (Mutually Recursive Methods)**.** The general form for the declaration of mutually recursive methods m of class A, and n of class B is the following.

$$\mathbf{meth}\ A\ m, B\ n = \mu\, X, Y \bullet \left(\ \mathit{pds}_m \bullet F(X, Y), \mathit{pds}_n \bullet G(X, Y)\ \right)$$

Mutual recursion is easily addressed in our theory. In this case, since m and n are mutually recursive, they are defined together, even though they are methods of different classes. This follows the standard approach to the definition of mutually recursive procedures. The vector of

programs m, n is defined as the least fixed point of the function from vectors of predicates to vectors of predicates defined by the bodies of m and n: $pds_m \bullet F(X, Y)$ and $pds_n \bullet G(X, Y)$. As an example, calling the methods m or n defined below, with a variable a as the result parameter, leads to the assignment of 0 to a.

$$\mathbf{meth}\ A\ m, B\ n = \mu\, X, Y \bullet \left(\begin{array}{l} \mathbf{val}\ x : \mathbb{Z};\ \mathbf{res}\ i : \mathbb{Z} \bullet i := x \lhd x = 0 \rhd Y(-x, i), \\ \mathbf{val}\ y : \mathbb{Z};\ \mathbf{res}\ j : \mathbb{Z} \bullet X(y-1, j) \lhd x > 0 \rhd X(y+1, j) \end{array} \right)$$

Once the recursion is resolved and the fixed-point operators are eliminated, the description of a multiple method declaration like **meth** $A\ m, B\ n = (pds_m \bullet mb), (pds_n \bullet nb)$ is a trivial extension of the definition of simple method declarations presented in Section 4.4. In many theories of object-orientation, mutual recursion is a difficulty. The complication is really attached to the fact that the mutually recursive methods may be declared in an independent way in separate classes. By splitting the block structure of a class into its basic semantic blocks, we trivially overcome this difficulty.

4.7.5 Method Call

Since we have already solved dynamic binding when dealing with the semantics of method declaration in Section 4.4.3, the semantics of method call is just a call to the value of the method. In other words, we have isolated the several aspects involved in a method call, so that dynamic binding is captured in the definition of the value of the method variable, which holds a parametrised program, and a method call is given mainly by the copy rule.

$$le.m(args) \mathrel{\widehat{=}} \mathbf{OO}(\{\mathcal{D}(le.m(args))\}_\perp;\ (pds_e \bullet p)(le, args))$$

$$\mathbf{where}\ m = \mathtt{pds}_e \bullet \mathtt{p}$$

Example 6. Suppose that $sc = sc^0$, $cls = cls^0$, and $atts = atts^0$, and after the declaration of classes, attributes and methods in the Examples 3 and 4, we have the program fragment below.

$$\begin{array}{l} \mathbf{var}\ a : Account; \\ a := \mathbf{new}\ BAccount; \\ a.credit(10) \end{array}$$

Due to dynamic binding, $a.credit(10)$ must execute the body of the method *credit* defined for the subclass *BAccount*. As described in Section 4.4, the value of *credit* is a conditional over the special variable named **self**. Below, we show how the program associated to the variable *credit* resolves the dynamic binding. The meaning of $a.credit(10)$ is defined in terms of $credit(a, 10)$, which we

consider below.

$$
\begin{array}{l}
\mathit{credit}(a, 10) \\
=\{ \text{ method expansion } \} \\
\left(\begin{array}{l} \textbf{valres self} : \textbf{\textit{Object}};\ \textbf{val}\ x : \mathbb{Z} \bullet \\ \quad \textbf{self}.\mathit{bonus} \ldots \lhd \textbf{self is}\ \mathit{BAccount} \rhd (\ldots \lhd \textbf{self is}\ \mathit{Account} \rhd \bot_{oo}) \end{array} \right)(a, 10) \\
\\
=\{ \text{ semantics of } \textbf{valres} \} \\
\textbf{var self} : \textbf{\textit{Object}}; \\
\quad \textbf{self} := a; \\
\quad \left(\begin{array}{l} \textbf{val}\ x : \mathbb{Z} \bullet \\ \quad \textbf{self}.\mathit{bonus} \ldots \lhd \textbf{self is}\ \mathit{BAccount} \rhd (\ldots \lhd \textbf{self is}\ \mathit{Account} \rhd \bot_{oo}) \end{array} \right)(10); \\
\quad a := \textbf{self}; \\
\textbf{end self} : \textbf{\textit{Object}} \\
\\
=\{ \text{ semantics of } \textbf{val} \} \\
\textbf{var self} : \textbf{\textit{Object}}; \\
\quad \textbf{self} := a; \\
\qquad \textbf{var}\ x : \mathbb{Z}; \\
\qquad\quad x := 10; \\
\qquad\quad \textbf{self}.\mathit{bonus} \ldots \lhd \textbf{self is}\ \mathit{BAccount} \rhd (\ldots \lhd \textbf{self is}\ \mathit{Account} \rhd \bot_{oo}); \\
\qquad \textbf{end}\ x : \mathbb{Z}; \\
\quad a := \textbf{self}; \\
\textbf{end self} : \textbf{\textit{Object}} \\
\\
=\{ \text{ as } \textbf{self}_t \text{ is } \mathit{BAccount}, \text{ the conditional reduces to its left branch } \} \\
\textbf{var self} : \textbf{\textit{Object}}; \\
\quad \textbf{self} := a; \\
\qquad \textbf{var}\ x : \mathbb{Z}; \\
\qquad\quad x := 10; \\
\qquad\quad \textbf{self}.\mathit{bonus} := \textbf{self}.\mathit{bonus} + 1; \\
\qquad\quad \textbf{self}.\mathit{balance} := \textbf{self}.\mathit{balance} + x; \\
\qquad \textbf{end}\ x : \mathbb{Z}; \\
\quad a := \textbf{self}; \\
\textbf{end self} : \textbf{\textit{Object}}
\end{array}
$$

This can be expanded to a predicate that establishes the final value of a to be its initial value with attributes updated by assignments. □

4.8 Conclusions

We have presented a stepwise introduction to object-oriented concepts in the Unifying Theories of Programming. We started with the definition of observational variables and healthiness conditions

(HCs), that restrict the values of these variables. The closedness properties of the HCs were stated and proved, to show, for example, that any two programs (or specifications) which are independently valid for a given HC can be combined by conjunction, disjunction or sequential composition, and the resulting program is still part of the lattice of predicates.

The declarations of classes, attributes and methods are defined in terms of the theory of designs, which itself filters the subset of terminating programs, combined with higher-order programming, which allows variables to record the behaviour associated to methods (abstractions), including dynamic binding resolution. We saw that each of these declarations preserves the defined HCs.

Type checking is an important issue in object-oriented languages. To record typing information, special variables were introduced and well-definedness conditions were specified for the new object-oriented commands and expressions. Those forms of commands and expressions already introduced by Hoare and He for a sequential programming language, however, had to be revised to cope with typing information, including method calls. We allow methods co- and contra-variance of arguments and return types.

We have seen also that the separation of declarations in different blocks has allowed the definition of (mutually) recursive methods in a straightforward manner. Furthermore, this has allowed a compositional approach which focuses on each feature in isolation. Another facility is specially related to dynamic binding resolution. When processing a method declaration, the observational variable responsible for that method is updated to reflect the new method's meaning. This might introduce a new observational variable for the method (when processing the first definition of a method), or updating the value of an existing variable to take into account the dynamic binding resulting from a method redefinition.

By all presented, we have a theory of object-orientation that handles inheritance, recursive data types, dynamic binding, polymorphism, and mutually recursive methods. Some considerations about verification are discussed in the sequel.

4.8.1 Verification

As we have already presented, the concept of refinement for the UTP is universal implication. If we are expected to introduce pre- and postconditions, in Hoare style, we have a straightforward manner to perform verification. As an example, suppose we have the following specification for the method *debit*: informally, we cannot perform a *debit* without enough funds.

$$\mathbf{pre}\ \mathbf{self}.balance > value$$
$$\mathbf{post}\ \mathbf{self}.balance' = \mathbf{self}.balance - value$$

To check this specification against its method definition, at any method call $le.m(args)$ the semantics could be extended to

$$\mathbf{OO}(\{\mathcal{D}(le.m(args)) \wedge \mathbf{pre}(args_c)\}_\perp;\ (pds_e \bullet p)(le, args);\ \mathbf{post}(args_r)),$$

where the precondition is extended with the copy-by-value variables (**val** and **valres**) and the postcondition is extended with the result variables (**res** and **valres**). Notice that, if the precondition is violated, the method call behaves like abort (**true**; $P \mathrel{\widehat{=}} \mathbf{true}$, from designs), and if the postcondition is violated it aborts the remaining program.

Moreover, there is a simpler alternative. Remember that in the UTP programs and specifications are interchangeable, the verifications thus would be part of the method body itself, and in this case the method call semantics remains the same. The expression

$$\mathbf{OO}(\{\mathcal{D}(le.m(args))\}_\perp;\ (pds_e \bullet \mathbf{pre};\ p;\ \mathbf{post})(le, args)),$$

is the same of declaring a lengthier method body with pre- and post-conditions.

Chapter 5

Pointers in the UTP

This chapter is concerned with how pointers are handled in formalisations of object-orientation and how it can be introduced in the UTP, firstly as a separated theory and them in a combined way. Section 5.1 provides an overview of pointers in the formalisms presented in Chapter 2, and the approach we follow. After that, Section 5.2 introduces the theory of pointers for UTP proposed by Cavalcanti, Harwood and Woodcock, with some minor revisions, and Section 5.3 discusses some ideas towards its integration with the theory of object-orientation presented in Chapter 4. Finally, Section 5.4 highlights our conclusions about the integration process.

5.1 Overview

It is well-known that most of the languages that are based on the imperative or the object-oriented paradigm in use today allow some sort of sharing. Some of them permit a more explicit manipulation of memory, like those where pointers can be directly handled, such as C or C++, and others are more restrictive like Java. In spite of its complexity, mainly due to the possibility of side-effects during memory manipulation, sharing is a design facility that cannot be excluded from programming language designs. It allows, for example, a better utilization of computational resources like memory, where objects can be recorded as graphs, instead of trees with replicated values; like CPU cycles, by avoiding cloning on assignments; or even the use of better practices of programming as described in design patterns, like Observers [GHJV95], where mutual references are required. As a consequence, the introduction of pointers in the theory of object-orientation is a natural step in its development. It surely allows more realistic programming languages modelling; on the other hand, however, it usually increases the complexity of the formalisation.

As presented in Section 2.1, there are many approaches [AL97, MPH97, PdB03, HLL06] that define object-orientation with a pointer semantics, and because of that, object creation, or manipulation, becomes more complex than we have presented in our theory so far. Notably, most

part of the works with sharing consider the existence of an object environment, where references and instances of objects are recorded, accessed or removed. A typical characterization, firstly introduced by Milner and Strachey [MS76], is a pair of functions to associate variable names to addresses and addresses to values, which can be primitive or objects formed by other references: $environment : name \mapsto address$ and $store : address \mapsto value$. Using pointers (aliasing) one needs also to be concerned more strictly with framed variables to avoid side-effects: unexpected changes in instances.

In general, such a characterisation requires a more detailed operational description of commands and expressions to show how the object environment is altered. Each of the approaches presented in Chapter 2, with the exception of ROOL that has a copy semantics, has its own set of variables to represent the object environment and rules to describe how it is affected by the constructions present in each formalism. The approach by He *et al.* presented in [HLL06], for example, considers a current configuration (a runtime environment) and uses it to describe refinement relations. Other approaches [AL97, MPH97, PHM98, PHM99, PdB03] use these object environments to define general Hoare-like axioms for object-oriented commands, or provide an operational semantics using these variables to establish a soundness theorem.

Our approach, presented in Chapter 4, which considers a copy semantics, assumes that there is an implicit object environment. In that case, objects are standard record values, and are treated in the same way as all the other kind of values in the UTP theories. When a new variable ($\mathbf{var}\ x : T$) is declared, it implicitly introduces a variable in the object environment (a non-hierarchical binding scheme), and allows the assignment of values or updates ($:=$), and its finalization ($\mathbf{end}\ x : T$). The set of variables present in this implicit environment is formed by (i) variables that hold objects, and also by (ii) variables capable of recording procedures (method meaning). It seems to be reasonable to consider the inclusion of a specific, and explicit, new set of observational variables to represent the object store as in He *et al.* [HLL06]. The approach adopted, however, abstracts from memory addresses. The next section shows that to handle sharing a detailed knowledge of the memory model implementation is not required; the main point is which objects share the same location.

Instead of altering the description of our theory for object-oriented programs, we choose to follow the design principles of the UTP, where theories are composed; we integrate our theory of object-orientation with an independent theory of pointers. We combine the two theories and thus define a subset of programs (predicates) that can be used to describe the behaviour of object-oriented programs with sharing. To do that we must consider new forms of (un)declaration of variable and object creation, among other concepts.

Another work that allows pointers introduction is Separation Logic [Rey02] which defines a model where heaps (object stores) can be divided in parts and specific logical operations can operate on these parts to locally state properties of the entire memory. Since it contains a different logic

background from the UTP, the characterisation of a theory for pointer and its integration with our theory for object-orientation would be more complex than integrating two theories of interest that use the UTP as a semantic framework. We are interested in the correct representation of features present in pointer and object-orientation theories, and its capacity to simplify the descriptions and proofs for object-oriented laws, with or without sharing. We are not concerned with the complexity of the memory model itself and its factoring.

5.2 Pointers Theory

A preliminary version of the theory summarised in this section has been published in [CHW06]. A more recent and detailed presentation is available as a technical report in [HCW07]; the focus of this work is on a theory for pointers in the presence of record values that can share fields. The authors define what they call a pointer machine to record sharing information and two forms of assignments are considered: value and pointer. All these operations are defined by means of predicates over observational variables that define a model for the pointer machine, in the UTP style.

The main difference between [HCW07] and [HLL05, HLL06] is the use of an implicit addressing scheme, more specifically, this theory is not concerned with addresses (as numbers) but with the valid variables, and whether they are sharing a given value or not. This strategy is inspired by the work of Brookes [Bro86] on an Algo-like language, and by Paige and Ostroff [PO04] with Eiffel. In the following subsections we present the observational variables, healthiness conditions, and operators of the pointer theory, before we consider its integration with our theory of object-orientation, which we discuss in the next section.

At the end of this section, in Example 7, a program using the theory allows us to analyse the dynamic behavior of the observational variables.

5.2.1 Observational Variables

To record pointers information, the pointer machine records addresses in use, the values associated to these addresses, and which ones identify the same location, and therefore are expected to have the same value. Addresses are represented by sequences of *Labels* separated or not by '.' such as x, $x.a$, y, $x.b$ and $x.a.c$. In the following definitions, for a set X, *seq* X stands for a set which contains all finite sequences of elements in X; thus, *seq Label* contains all finite sequences of labels[1]. The set of finite addresses Ad excludes the empty sequence of labels.

$$Ad \mathrel{\widehat{=}} (seq\ Label) \backslash \{<>\}$$

[1] In [HCW07], the hypothetical machine can address finite or infinite non-empty sequences of labels, for simplicity we consider only finite addresses. This simplification is reflected in some forthcoming definitions.

The observational variables A, V, and S that model the pointer machine form a triple:

$$< A : \mathbb{P} Ad, V : Ad \mapsto Value, S : Ad \leftrightarrow Ad >$$

where A is the set of valid addresses, V is a partial function from addresses to values, and S is an equivalence relation that associates addresses that share a location. The set *Value* include primitives or records. In other words, if an address is in A, then it is a valid variable (or field) access, V represents the *store* from the previous approaches; the *environment* has not a direct representation. A preliminary analysis reveals some relations between these three elements.

- dom $V \subseteq A$: only valid addresses have values;
- $S \subseteq A \times A$: the equivalence relation S must be a subset of the cartesian product of valid addresses, in other words, only valid addresses can be shared;
- $S = S^*$: the S relation is the reflexive, symmetric and transitive, that is, all addresses that are shared are associated to all their peers and vice-versa.

Next section shows that these restrictions are imposed by healthiness conditions.

Equality In this theory, two kinds of equality of values are considered: address test, denoted as $=_p$, and value test $=_v$. To define this new equality, a projection $< A.p, V.p >$ of the machine address for the variables tested is considered. The projection of A

$$A.p \mathrel{\widehat{=}} \{q : Ad \mid p.q \in A\}$$

filters all addresses suffixes that have the same prefix p. For example, if $A = \{x, y, x.a, y.b, x.d, x.d.c\}$ then $A.x$ yields $\{a, d, d.c\}$. The projection of V

$$V.p \mathrel{\widehat{=}} \{q : Ad \mid p.q \in \text{dom } V \bullet q \mapsto V(p.q)\}$$

filters all values of the variables prefixed by p, which we call children of p. For example, if $V = \{x.a \mapsto v_1, y.b \mapsto v_2, x.d.c \mapsto v_3\}$ then $V.x$ yields $\{a \mapsto v_1, d.c \mapsto v_3\}$. The values of two variables (or field accesses) p and q are equivalent if the projections of these variables for the sets A and V are equal.

$$p =_v q \equiv A.p = A.q \land V.p = V.q$$

The pointer test is simpler; two addresses point to the same value if the pair (p, q) belongs to the equivalence S.

$$p =_p q \equiv (p, q) \in S$$

5.2.2 Healthiness Conditions

This section introduces the HCs of this theory. During their presentation, other necessary definitions are explained and used. The first one is the concept of a prefix: $x < y$ is used to denote that x is a prefix of y, where x is an address. For example, x is a prefix for $x.a$ and $x.a.c$, and $x.a$ is a prefix for $x.a.c$. The first HC says that A is prefix closed, that is, all valid prefixes for addresses in A must be in A, otherwise we could have a sequence of labels which does not exist. i.e. $x.a \in A$ and $x \notin A$.

HP1 $P = P \land \forall\, a_1 : A;\ a_2 : Ad \mid a_2 < a_1 \bullet a_2 \in A$

Another interesting concept is that of terminal nodes; addresses in A which are not prefixes. The function $term(X)$ filters all those sequences of labels in X which are terminal. It is defined as

$$term(X) \mathrel{\widehat{=}} \{x : X \mid \neg\, \exists\, y : X \bullet x < y\}$$

The second HC says that only terminal nodes have values.

HP2 $P = P \land \mathrm{dom}\ V = term(A)$

The next healthiness condition connects the variables in the alphabet to those in the pointer machine. A new notation $'x$ is used to refer to the name of x in the pointer machine. The variables of the alphabet are the first elements of the addresses in the set A; for filtering these names a function

$$vars(X) \mathrel{\widehat{=}} \{x : X \bullet x(1)\}$$

is defined. For example, the result of $vars(\{x, y, x.a, y.b, x.d, x.d.c\})$ is $\{x, y\}$. The next step is to de-reference the variables to recover its values; for every variable which is in the set of programming variables a new function '!' is used (A, V and S are not in this set). If the address x is a terminal node, the result is the value itself, $V(x)$, otherwise it is a projection of V. The definition of '!' is similar for initial or final variables, as presented below.

$$!x \mathrel{\widehat{=}} \begin{cases} V(x)\,,\ \textit{if}\ x \in term(A) \\ V.x\ \ ,\ \textit{otherwise.} \end{cases} \qquad !x' \mathrel{\widehat{=}} \begin{cases} V'(x)\,,\ \textit{if}\ x \in term(A') \\ V'.x\ \ ,\ \textit{otherwise.} \end{cases}$$

Thus, variables are related to pointer machine entries[2] as follows. Where NPV stands for non programming variables: the set of variables which are not present in the pointer machine.

HP3 $P = P \land \bigwedge\{\, 'x : vars(A) \bullet x = !'x\}$

where $NPV \cap vars(A) = \emptyset$

[2]The original definition of **HP3** presented in [HCW07] is too restrictive to be used in theories integration. There $NPV = in\alpha(P) \setminus \{A, V, S\}$, which requires other observational variables to be in the pointer machine, and this is not truth.

and $\{A, V, S\} \subseteq NPV$.

In theory extension (or integration) NPV may be larger. The next HC establishes that the elements in S must belong to A and S is the reflexive, symmetric and a transitive closure of itself.

HP4 $P = P \land S \in (A \leftrightarrow A) \land S = S^*$

The fifth healthiness condition relates all suffixes to their prefixes; in other words, if two different address prefixes (x, y) that share a value ($(x, y) \in S$) have the same suffixes $(x.a, y.a)$, then these suffixes also share the same values ($(x.a, y.a) \in S$).

HP5 $P = P \land fclos_A\ S$

where $fclos_A\ E \mathrel{\widehat{=}} \forall\, x, y : Ad \mid (x, y) \in E \bullet \forall\, a : Ad \mid x.a \in A \land y.a \in A \bullet (x.a, y.a) \in E$

Finally, the sixth HC guarantees that if two terminals share the same location then their values are equal.

HP6 $P = P \land \forall\, a, b : Ad \bullet (a, b) \in S \land a \in \text{dom}\ \ V \Rightarrow b \in \text{dom}\ V \land\ V(a) = V(b)$

The corresponding HCs for terminal variables are named **HP7-12**, and are similar to these replacing the initial variables by final variables. For example, $\mathbf{HP11}(P) = P \land fclos_{A'}\ S'$. **HPI** is defined as $\mathbf{HP1} \circ \mathbf{HP2} \circ \mathbf{HP3} \circ \mathbf{HP4} \circ \mathbf{HP5} \circ \mathbf{HP6}$ and restricts initial values only. For the composition of **HP1-12** we refer as **HP**.

Conjunctive Healthiness Conditions

An important result provided by the work of Harwood *et al.* is the characterisation of conjunctive healthiness conditions (**CH**). HCs that are described in terms of conjunctions

$\mathbf{CH}(P) = P \land \psi$

where ψ is a predicate. This general characterisation allows the study of general properties of such kind of HCs, in special that **CH** are closed with respect to conjunction, disjunction, conditional, composition and recursion. In other words, once a HC is defined as conjunction, it is granted that it is closed under this set of operators. This result could be used, for example, to justify closeness of operators introduced in Chapter 4 and proved in Appendix B.

Another particularity of this kind of healthiness conditions, not observed in [HCW07], is that if two HCs are conjunctive, let say $\mathbf{CH}_1$ and $\mathbf{CH}_2$, they also commute, as proved below.

Law 1. <**CH***s-commutativity*>

$\mathbf{CH}_1 \circ \mathbf{CH}_2 = \mathbf{CH}_2 \circ \mathbf{CH}_1$

Proof.

$\mathbf{CH}_1 \circ \mathbf{CH}_2(P)$ *[definition of* $\mathbf{CH}$*s, composition]*
$= (P \wedge \psi_2) \wedge \psi_1$ *[propositional logic]*
$= (P \wedge \psi_1) \wedge \psi_2$ *[definition of* $\mathbf{CH}$*s, composition]*
$\mathbf{CH}_2 \circ \mathbf{CH}_1(P)$ □

5.2.3 Variables

The introduction of a new variable has to be reflected in the pointer machine. The following definition shows the declaration of a variable x includes it as a valid address name, associate it to an arbitrary value, and relates the variable to itself.

$$\mathbf{var}\ x \mathrel{\widehat{=}} \mathbf{HPI} \circ \mathbf{HP9} \left(\exists\, v : \mathit{Value} \bullet \begin{pmatrix} A' = A \cup \{x\} \wedge \\ V' = V \oplus \{x \mapsto v\} \wedge \\ S' = S \cup \{x \mapsto x\} \end{pmatrix} \right)$$

$$\mathbf{provided}\ x \notin A$$

This definition does not pose any restrictions on the possible values associated to a given variable. We have seen that restricting the values taken by variables is essential to the correct behaviour of object-oriented programs, therefore we have to review this definition when addressing integration.

The finalization of a variable x implies in its removal from A, V, and S. The removal, however, must consider that all addresses with that prefix, denoted as '$x\infty$', also become invalid, and are expected to be removed as well.

$$\mathbf{end}\ x \mathrel{\widehat{=}} \mathbf{HPI} \circ \mathbf{HP9} \begin{pmatrix} A' = A \setminus \{x\infty \cup \{x\}\} \wedge \\ V' = (x\infty \cup \{x\}) \lhd V \wedge \\ S' = (x\infty \cup \{x\}) \lhd S \rhd (x\infty \cup \{x\}) \end{pmatrix}$$

$$\mathbf{provided}\ x \in A$$
$$\mathbf{where}\ x\infty \mathrel{\widehat{=}} \{a : Ad \bullet x.a\}$$

In this definition, the Z notation [Spi92] for domain and image anti restrictions are used. These functions filter a relation considering elements in domain and image. For example, the V' mapping does not contain x or elements prefixed by x in its domain.

5.2.4 Commands

When dealing with pointers one can associate values to variables or make variables to share locations (aliasing). Thus, the theory of pointers has two kinds of assignments: an assignment of values denoted as ':=', expected to change variable values recorded in V, and pointer assignments ':–', expected to associate two addresses to the same value. The complexity of these commands

arise when non-terminal variables are assigned; in this case more complex adjustment have to be performed in the variables A, V and S to keep their consistency.

It is important to emphasise that this theory does not handle types. All variations of assignments are possible between variables and values. Compatibility between variable types and their assigned values is not checked, this will be our first step into the goal of integration, associate types to values.

Before defining the assignments, a function $share_S(x)$ is introduced to recover all variables in the equivalence relation S that have their values shared with x. Its general form is described below, where X is an equivalence relation.

$$share_X(x) \mathrel{\widehat=} X(\!|\ \{x\}\ |\!)$$

Assignments of values can be split into two cases: the address is terminal ($x\infty$ is empty) or is an internal node. The case of terminal nodes $:=_t$ is simpler; the expected result is the association of a new value e to all those addresses that share the value with x, and the other variables remain unchanged.

$$x :=_t e \mathrel{\widehat=} \mathbf{HPI} \circ \mathbf{HP9} \begin{pmatrix} A' = A \wedge \\ V' = V \oplus \{a : share_S(x) \bullet a \mapsto e\} \wedge \\ S' = S \end{pmatrix}$$

provided $x \in \mathrm{dom}\ V$

The second type of value assignment is that of non-terminal addresses[3]. In its definition the update of the prefixed addresses is also required. That is, in $x :=_i e$ all variables that have prefix x (i.e. $x.a, x.d, x.d.c$) are expected to be updated appropriately. Two auxiliary functions are defined to filter all addresses of interest. The function $X^\uparrow$ yields all addresses that have prefixes in X, and $ext_S(x)$ selects all address names that are prefixed by x or addresses that share their values with x.

$$X^\uparrow \mathrel{\widehat=} \bigcup\{x : X \bullet x\infty\}$$
$$ext_X(x) \mathrel{\widehat=} share_X(x)^\uparrow$$

All extensions of x are removed from A, all values associated to children (addresses with prefix x) of x have their values removed and a new association to the value of e is considered, and at last, all relations between these children are also removed from the sharing equivalence S. The set $share_S(x) \cup ext_S(x)$ is the set of all variables that share values with x and all their extensions, and $share_S(x) \setminus ext_S(x)$ selects only those addresses which share values with x excluding their children

[3]This kind of assignment exist in this theory because of its absence of typing restrictions, in the object-oriented theory assignments of values to non-terminals are restricted to **null** assignments.

that became inexistent after this assignment.

$$x :=_i e \mathrel{\widehat{=}} \mathbf{HPI} \circ \mathbf{HP9} \begin{pmatrix} A' = A \setminus ext_S(x) \wedge \\ V' = (ext_S(x) \lhd V) \cup \{a : (share_S(x) \setminus ext_S(x)) \bullet a \mapsto e\} \wedge \\ S' = ext_S(x) \lhd S \rhd ext_S(x) \end{pmatrix}$$

provided $x \in A, x \notin \mathrm{dom}\ V$

The second type of assignment is that of pointers. Again, a distinction between terminal and internal nodes is made. For terminal nodes, the set of valid addresses prefixed by x is extended with all the suffixes of addresses prefixed by y, values previously associated to x are removed and the values associated with the new suffixes of y are associated to the new addresses included in A, and finally, for S, the original relations of x are replaced by the relation of x with y and its forward closure.

$$x :-_t y \mathrel{\widehat{=}} \mathbf{HPI} \circ \mathbf{HP9} \begin{pmatrix} A' = A \cup \{a : Ad \mid y.a \in A \bullet x.a\} \wedge \\ V' = (\{x\} \lhd V) \oplus \{a : \mathrm{dom}\ V;\ fa : seq\ Label \mid a = y.fa \bullet x.fa \mapsto V(y.fa)\} \wedge \\ S' = ((((\{x\} \lhd S \rhd \{x\}) \cup \{x \mapsto y\}) \cup \{a : Ad \mid y.a \in A \bullet x.a \mapsto y.a\})^* \end{pmatrix}$$

provided $x \in \mathrm{dom}\ V$

The internal case is a generalization of the terminal case, where all addresses with prefix x ($x\infty$) are expected to be removed from A, V and S before the required adjustments.

$$x :-_i y \mathrel{\widehat{=}} \mathbf{HPI} \circ \mathbf{HP9} \begin{pmatrix} A' = (A \setminus x\infty) \cup \{a : Ad \mid y.a \in A \bullet x.a\} \wedge \\ V' = (x\infty \lhd V) \oplus \{a : \mathrm{dom}\ V;\ fa : seq\ Label \mid a = y.fa \bullet x.fa \mapsto V(y.fa)\} \wedge \\ S' = (((x\infty \cup \{x\}) \lhd S \rhd (x\infty \cup \{x\})) \cup \{x \mapsto y\} \cup \\ \qquad \{a : Ad \mid y.a \in A \bullet x.a \mapsto y.a\})^* \end{pmatrix}$$

provided $x \in A, x \notin \mathrm{dom}\ V$

In [HCW07] these kinds of assignments are proved to be **HP7-12** healthy, and moreover theorems and laws, such as $share_{(ext_S(x) \lhd S \rhd ext_S(x))}(x) = share_S(x) \setminus ext_S(x)$ for finite addresses, are described and proved.

5.2.5 Records

The last concept relevant to our integration is record creation. It defines what are the effects on the pointer machine of allocating a new memory space. Different from [HCW07] we consider $\mathbf{new}(f)$ as an expression whose value is defined as a mapping from names to values as defined below, where f stand for a set of names for record fields.

$$\mathbf{new}(f) \mathrel{\widehat{=}} \{n : name;\ v : Value \mid n \in f \bullet n \mapsto v\}$$

The assignment of a $\mathbf{new}(f)$ expression to x, '$x :=_r \mathbf{new}(f)$', is thus defined as

$$x :=_r \mathbf{new}(f) \mathrel{\widehat{=}} \mathbf{HPI} \circ \mathbf{HP9} \begin{pmatrix} A' = (A \setminus x\infty) \cup \{n : name \mid n \in \mathrm{dom}\, map \bullet x.n\} \wedge \\ V' = ((x\infty \cup \{x\}) \lhd V) \oplus \{n : name \mid n \in \mathrm{dom}\, map \bullet x.n \mapsto map(n)\} \wedge \\ S' = (x\infty \cup \{x\}) \lhd S \rhd (x\infty \cup \{x\}) \end{pmatrix}$$

provided $x \in A$
where $map = \mathbf{new}(f)$

With a detailed analysis it is possible, and not surprising, to verify that there is a close relation with the definition of variable declaration:

- in declarations x is introduced in A; record creation associates the record fields $f_0, \ldots, f_n$ to addresses prefixed by x in A $(x.f_0, \ldots, x.f_n)$;
- in declarations x is associated to an arbitrary value, and in record creation the values previously associated to x and all addresses with prefix x are removed from V and the new record fields values are included[4];
- in declarations x shares a value with itself, after record creation all new addresses $x.f_0, \ldots, x.f_n$ are fresh and x itself has not a value, therefore must be removed from S.

The record creation is valid for any set of names defined in f, but the record assignment is defined only for those addresses in A.

Example 7 (Pointer theory)**.** This example shows how the pointer machine behaves for a simple program presented in Figure 5.1. Considering A, V, and S initially empty, the effect of program

l_1: **var** x;
l_2: **var** y;
l_3: $x := \mathbf{new}(a, b)$;
l_4: $y :\!- x$;
l_5: $y := \mathbf{new}(c, d)$;
l_6: $y.c :\!- x$;
l_7: $y.c.a := v_7$;
l_8: $x := v_8$;
l_9: **end** x;
l_{10}: **end** y

Figure 5.1: A program using pointers.

execution is presented in the sequel, where we describe the values of the observational variables

[4] In the context of object-orientation it could be seen as a variable previously with **null**, and after object creation (**new**) an instance is associated.

after each command. The values from v_1 to v_8 are arbitrary, but we assume that v_7 and v_8 are primitives. The assignment types ($:=_t$, $:=_i$, or $:=_r$) are inferred from the context where they are used.

{ just before starting all variables are empty }
$A = \{\} \wedge V = \{\} \wedge S = \{\}$

l_1, l_2 :‘**var** x; **var** y’ { variable declaration }
$A' = \left\{x, y\right\} \wedge V' = \left\{x \mapsto v_1, y \mapsto v_2\right\} \wedge S' = \left\{x \mapsto x, y \mapsto y\right\}$

l_3 :‘$x :=$ **new**(a, b)’ { record creation and assignment ($:=_r$) for variable x }
$A' = \begin{Bmatrix} x, x.a, x.b, \\ y \end{Bmatrix} \wedge V' = \begin{Bmatrix} x.a \mapsto v_3, x.b \mapsto v_4, \\ y \mapsto v_2 \end{Bmatrix} \wedge S' = \left\{x \mapsto x, y \mapsto y\right\}$

l_4 :‘$y :\!- x$’ { pointer assignment, associate a chain of references ($:\!-_i$) }
$A' = \begin{Bmatrix} x, x.a, x.b, \\ y, y.a, y.b \end{Bmatrix} \wedge V' = \begin{Bmatrix} x.a \mapsto v_3, x.b \mapsto v_4, \\ y.a \mapsto v_3, y.b \mapsto v_4 \end{Bmatrix} \wedge S' = \begin{Bmatrix} x \mapsto x, y \mapsto y, \\ x \mapsto y, \\ x.a \mapsto y.a, x.b \mapsto y.b \end{Bmatrix}^*$

l_5 :‘$y :=$ **new**(c, d)’ $\left\{ \begin{array}{l} \text{record creation and assignment } (:=_r), \\ \text{creates new values for } y.c \ldots \text{ and undo references to } x \end{array} \right\}$
$A' = \begin{Bmatrix} x, x.a, x.b, \\ y, y.c, y.d \end{Bmatrix} \wedge V' = \begin{Bmatrix} x.a \mapsto v_3, x.b \mapsto v_4, \\ y.c \mapsto v_5, y.d \mapsto v_6 \end{Bmatrix} \wedge S' = \left\{x \mapsto x, y \mapsto y\right\}$

l_6 :‘$y.c :\!- x$’ { pointer assignment, of a terminal address $y.c$ to a non-terminal x ($:\!-_i$) }
$A' = \begin{Bmatrix} x, x.a, x.b, \\ y, y.c, y.d, \\ y.c.a, y.c.b \end{Bmatrix} \wedge V' = \begin{Bmatrix} x.a \mapsto v_3, x.b \mapsto v_4, \\ y.d \mapsto v_6, \\ y.c.a \mapsto v_3, y.c.b \mapsto v_4 \end{Bmatrix} \wedge S' = \begin{Bmatrix} x \mapsto x, y \mapsto y, \\ y.c \mapsto x, \\ y.c.a \mapsto x.a, \\ y.c.b \mapsto x.b \end{Bmatrix}^*$

l_7 :‘$y.c.a := v_7$’ { value assignment, to a terminal address which shares a value ($:=_t$) }
$A' = \begin{Bmatrix} x, x.a, x.b, \\ y, y.c, y.d, \\ y.c.a, y.c.b \end{Bmatrix} \wedge V' = \begin{Bmatrix} x.a \mapsto v_7, x.b \mapsto v_4, \\ y.d \mapsto v_6, \\ y.c.a \mapsto v_7, y.c.b \mapsto v_4 \end{Bmatrix} \wedge S' = \begin{Bmatrix} x \mapsto x, y \mapsto y, \\ y.c \mapsto x, \\ y.c.a \mapsto x.a, \\ y.c.b \mapsto x.b \end{Bmatrix}^*$

l_8 :‘$x := v_8$’ { value assignment, to a non-terminal address ($:=_i$) }

$$A' = \left\{ \begin{matrix} x, \\ y, y.c, y.d \end{matrix} \right\} \wedge V' = \left\{ \begin{matrix} x \mapsto v_8, \\ y.c \mapsto v_8, \\ y.d \mapsto v_6 \end{matrix} \right\} \wedge S' = \left\{ \begin{matrix} x \mapsto x, y \mapsto y, \\ y.c \mapsto x \end{matrix} \right\}^*$$

l_9 :‘**end** x’{ undeclaration }

$$A' = \left\{ y, y.c, y.d \right\} \wedge V' = \left\{ \begin{matrix} y.c \mapsto v_8, \\ y.d \mapsto v_6 \end{matrix} \right\} \wedge S' = \left\{ y \mapsto y \right\}$$

l_{10} :‘**end** y’{ undeclaration }

$A' = \{\} \wedge V' = \{\} \wedge S' = \{\}$

Observing this example we conclude that to consider object instances instead of records some changes are required, as the use of class information to select the appropriate fields of a record. Moreover, the assignment of line 8 is not a valid assignment for typed languages. These are some of the topics handled in the next section. □

5.3 Integration

The objective of this section is to relate concepts present in both theories under a combined one where all previous observational variables and healthiness conditions are considered. That is, we consider all variables already presented; all definitions from object-orientation are inherited and some definitions are progressively changed to be characterised in terms of the pointer machine. For example, record creation is related to object creation and some adjustments are required for a combined theory, i.e. record creation of pointers changes to use and record typing information required by object-orientation.

The set of predicates that satisfy this new hybrid theory lies in the intersection of predicated delimited by **OO** and **HP**. Moreover, during the integration process some extra information is required and a new observational variable is used; this new observational variable uses information of both theories and its restrictions are provided by healthiness conditions identified by **PO**. In the following sections we present the changes required for the integration; in the Appendix D all definitions and constructs of the integrated theory are presented.

5.3.1 Observational Variables and HCs

The following observational variable is responsible for recording the dynamic type of addresses; given an address, which has a value, it is possible to recover its dynamic type. This information is

required for the integrated definition of commands such as assignments, and expressions such as type tests.

Definition 14 (Dynamic types)**.** The type of a value associated to an address is recorded in $dts : Ad \mapsto Type$.

A possible value for this variable is dts^0 defined below, which is useful in examples.

$$dts^0 = \{\}$$

There is a close relationship between this new observational variable and some variables present in the theories of object-orientation and pointer. We represent this relationship as a new healthiness condition which establishes that all valid addresses have a dynamic type in *dts*, and all dynamic types associated to a given address are present in *cls* or are primitives.

PO1 $P = P \land \text{dom}\, dts = A \land \text{ran}\, dts \subseteq Type$

PO2 stands for the restriction of final values.

PO2 $P = P \land \text{dom}\, dts' = A' \land \text{ran}\, dts' \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup cls'$

As usual, **PO** is used to refer to their composition **PO1** $\circ$ **PO2**. Since **PO1** is a conjunctive healthiness condition (**CH**) the results of Harwood *et al.* with respect to closeness of operators, and our law for commutativity, can be applied.

All definitions in the integrated theory are supposed to be **OO**, **HP** and **PO** healthy; therefore, they are all surrounded by **IT**, defined as:

$$\mathbf{IT} \mathrel{\widehat{=}} \mathbf{OO} \circ \mathbf{HP} \circ \mathbf{PO}$$

An important remark is that according to **Law** 1<**CH***s-commutativity*>, the order of application of **HP**, **OO** and **PO** is irrelevant for this definition, and in the forthcoming ones.

5.3.2 Restricting HP3

In a perfect integration, the healthiness conditions of a theory should not interfere with the HCs from the other one. Unfortunately this is not the case, in order to allow integration we have to extend the set of variables considered for *NPV* presented in **HP3**. Observe that the theory of object-orientation has other observational variables *cls*, *sc* and *atts*, and method variables, which are not programming variables and cannot be de-referenced in the pointer machine. Thus, in the integration we have that

$$\{A, V, S, cls, sc, atts, dts\} \cup M \subseteq NPV$$

where : M is the set of method variables.

Only variables outside this set can be de-referenced; from **HP3** we have that $NPV \not\subseteq vars(A)$. Since we are restricting the set of variables for NPV, we are strengthening the condition associated to **HP3**, therefore, the set of valid predicates for this new restriction is smaller than before

$$\{A, V, S, cls, sc, atts, dts\} \cup M \subseteq NPV \Rightarrow \{A, V, S\} \subseteq NPV$$

In other words, the set of predicates filtered by this condition is smaller than, and a subset of, those filtered by the original version of **HP3**.

5.3.3 Variables

In the theory of pointers, variable information is recorded in the pointer machine. We reuse the definition of object-orientation where the type has to be previously defined, and we extended it to consider *dts*. The definition had to change to accommodate a new proviso in the design ($x \notin A$), and the postcondition had to include all changes required in *dts*, A, V and S. All other variables not mentioned remain unchanged ($w' = w$).

$$\mathbf{var}\ x : T \mathrel{\widehat{=}} \mathbf{IT} \left(\begin{array}{l} \{T \in Type\}_\perp;\ \mathbf{var}\ xt, x;\ x \notin A \vdash \exists\, v : Value \bullet \\ \\ \\ \\ \\ \\ \end{array} \begin{pmatrix} xt' = T \land \\ x' \in \mathcal{V}(T) \land \\ A' = A \cup \{x\} \land \\ V' = V \oplus \{x \mapsto v\} \land \\ S' = S \cup \{x \mapsto x\} \land \\ dts' = dts \oplus \{x \mapsto T\} \land \\ w' = w \end{pmatrix} \right)$$

provided $x \notin in\alpha(\mathbf{var}\ x : T)$
where $w \in in\alpha(\mathbf{var}\ x : T) \setminus \{x, xt, A, V, S, dts\}$

In this definition the dynamic type of x is recorded in *dts*, and the pointer machine variables are updated according to the pointer machine definition for variable declaration. Only typed variables are recorded in A; method variables receive the same treatment as in object-orientation theory. The variable removal is a simple merge of both theory definitions, with adjusts.

$$\mathbf{end}\ x : T \mathrel{\widehat{=}} \mathbf{IT} \left(\left(x \in A \vdash \begin{pmatrix} A' = A \setminus \{x\infty \cup \{x\}\} \land \\ V' = (x\infty \cup \{x\}) \lhd V \land \\ S' = (x\infty \cup \{x\}) \lhd S \rhd (x\infty \cup \{x\}) \land \\ dts' = (x\infty \cup \{x\}) \lhd dts \land \\ w' = w \end{pmatrix} \right);\ \mathbf{end}\ x, xt \right)$$

provided $x \in in\alpha(\mathbf{end}\ x : T)$
where $w \in in\alpha(\mathbf{end}\ x : T) \setminus \{x, A, dts, V, S\}$

In this removal, since x is excluded of the pointer machine, the x' becomes undefined, therefore, it is not part of w.

5.3.4 What is a *Value*?

We have seen that the pointer theory records values in a pointer machine; those values recorded by pointer machines are generally referred as elements of the set *Value*. For this combined theory, *Value* is defined as

$$Value \mathrel{\widehat=} \{ T : Type;\ i : name \mid i \in T \bullet i \}$$

Considering each type in isolation: if $T = \mathbb{Z}$, we have $\{-\infty, \ldots, 0, \ldots, +\infty\}$; if $T = \mathbb{B}$, we have $\{\mathbf{true}, \mathbf{false}\}$; the elements for $T \in cls$, however, are **null**. Thus, *Value* represents the set

$$\{-\infty, \ldots, 0, \ldots, +\infty\} \cup \{\mathbf{true}, \mathbf{false}\} \cup \{\mathbf{null}\}$$

5.3.5 Expressions

Since types are important in the definitions of valid assignments and other operations in object-orientation, we have to characterize what are the values yielded by expressions like 9999 (a primitive value) or x (an address) in the integrated theory.

In the theory for object-orientation the result yielded by an expression is formed by type and value (e_t, e_v) which is used to perform type checking in the program. If a constant is used, for example 9999, it implicitly carries type information $(\mathbb{Z}, 9999)$; for objects the same happens. If x records an object, the value for x is characterised as a pair (x_t, x_v), where x_t is the dynamic type for x and x_v is **null**, or a mapping from attribute names to values, which are also pairs.

In the pointer theory, the types are not considered. The result of an expression is a *Value*, for primitive types, or the result of the de-referencing function '!', for variables recorded into the pointer machine.

The challenge is how to connect both theories in such a way that the HCs of the object-orientation theory and the HCs of the pointer theory hold. In the object-orientation theory, the HCs **OO** do not restrict the values associated to variables, but its formalization rely on type information associated to values to validate assignments and other constructions. For pointers, however, **HP3** establishes that the value of a variable is the result of the de-referencing function '!'.

To satisfy these two restrictions we introduce type information in the interpretation of $!x$. As already said, the typing information is required for the correct behavior of object-oriented programs. It is the reason for a new variable *dts* which records type information associated to addresses. The dynamic type of a variable is recorded in *dts*, and the value in V.

In the integrated theory, if an expression is a constant value as 9999, it still stands for $(\mathbb{Z}, 9999)$. In other words, we have implicit type information. For a variable x we have that the value yielded

by its de-referencing is also a pair defined as follows.

$$!x \mathrel{\widehat=} \begin{cases} (dts(x), V(x)) \,, & \textit{if } x \in \textit{term}(A) \\ (dts(x), V.x) & \textit{, otherwise.} \end{cases} \qquad !x' \mathrel{\widehat=} \begin{cases} (dts(x'), V'(x)) \,, & \textit{if } x \in \textit{term}(A') \\ (dts(x'), V'.x) & \textit{, otherwise.} \end{cases}$$

The value of a variable is a pair with its dynamic type, and its de-referencing in the pointer machine. For example, in a context where

$$dts = \begin{Bmatrix} x \mapsto \mathbb{Z}, \\ y \mapsto \mathbb{B} \end{Bmatrix}, A = \begin{Bmatrix} x, \\ y \end{Bmatrix}, V = \begin{Bmatrix} x \mapsto 1, \\ y \mapsto \textbf{false} \end{Bmatrix}$$

the values of x and y are respectively $(\mathbb{Z}, 1)$ and $(\mathbb{B}, \textbf{false})$. In a context

$$dts = \left\{ \begin{array}{l} x \mapsto \textit{Account}, \\ x.\textit{number} \mapsto \mathbb{Z}, \\ x.\textit{balance} \mapsto \mathbb{Z}, \\ x.\textit{contact} \mapsto \textit{Contact} \end{array} \right\}, A = \left\{ \begin{array}{l} x, \\ x.\textit{number}, \\ x.\textit{balance}, \\ x.\textit{contact} \end{array} \right\}, V = \left\{ \begin{array}{l} x.\textit{number} \mapsto 1234, \\ x.\textit{balance} \mapsto 1, \\ x.\textit{contact} \mapsto \textbf{null} \end{array} \right\}$$

the value of x is $(\textit{Account}, \{\textit{number} \mapsto 1234, \textit{balance} \mapsto 1, \textit{contact} \mapsto \textbf{null}\})$. The value of $x.\textit{contact}$ is $(\textit{Contact}, \textbf{null})$.

Well-definedness

In integrated theory well-definedness conditions described for object-orientation are still valid with the exception of variables rules. For a variable be valid it must also belong to A.

Variables

$$\begin{array}{rcl} \mathcal{D}(x) & \mathrel{\widehat=} & xt \in \textit{Type} \land x \in A \\ \mathcal{D}(\textbf{self}) & \mathrel{\widehat=} & \textbf{self}t \in \textit{cls} \land \textbf{self} \in A \end{array}$$

Object Creation, Type Test, Type Cast and Variable Accesses

The definition of object creation reflects the object-orientation type restriction for attributes and add type information for the register. The result of **new**, is a constant structure; changes are not allowed until this structure is recorded in the pointer machine.

$$\textbf{new}\ N \mathrel{\widehat=} (N, \{n : \textit{name};\ v : \textit{Value} \mid n \in \operatorname{dom} \textit{map} \land v \in \textit{map}(n) \bullet n \mapsto v\})$$

$$\textbf{where}\ \textit{map} = \mathcal{U}(\textit{atts}, \textit{cls}, N)$$

Values assigned to attributes are in *Value*: an integer, a boolean or an object value, and the well-definedness for **new** N is preserved as $N \in \textit{cls}$. Moreover, the **new** declaration yields the same representation of an object as previously argued for a variable access (a pair of a type and a mapping of attribute to values).

In the definition for assignment of **new** values, this structure is recorded in the pointer machine as records used to be in the pointer theory, but extra information about attribute types are also recorded in *dts* with the help of $\mathcal{U}(atts, cls, N)$.

The type test and type cast are resolved in the same manner as in the theory of object-orientation; the variable and attribute accesses, however, are much simpler, $!x$ and $!le.x$, respectively.

Equality The definition of equality of values in the theory of pointers must be changed to

$$p =_v q \equiv A.p = A.q \wedge !p =!q$$

since the previous one could not be used to compare two addresses like x and y whose values were primitives, for example. The value of an address p is provided by the '!' function, if it holds a primitive or an object value. The comparison of addresses is the same as that of the pointer theory.

5.3.6 Commands

This section shows how command definitions in the theory for object-orientation have to change to accommodate the pointer machine.

Well-definedness

All well-definedness rules, with the exception of method call, remain unchanged. For example, the definition of well-definedness for assignments

Assignment to variables
$\mathcal{D}(x := e) \;\widehat{=}\; \mathcal{D}(x) \wedge \mathcal{D}(e) \wedge e_t \preceq xt$

Assignment to attributes
$\mathcal{D}(le.x := e) \;\widehat{=}\; \mathcal{D}(le.x) \wedge \mathcal{D}(e) \wedge e_t \preceq \mathcal{U}(atts, sc, le_t)(x)$

remain unchanged, but $\mathcal{D}(x)$ includes $x \in A$, and for $\mathcal{D}(le.x)$ the first label in $le.x$ must be in A.

As we have another type of assignment $(:-)$, we describe its well-definedness rules below, which are similar to assignment of values.

$$\mathcal{D}(x :- y) \mathrel{\widehat{=}} \mathcal{D}(x) \wedge \mathcal{D}(y) \wedge y_t \preceq xt$$
$$\mathcal{D}(le.x :- y) \mathrel{\widehat{=}} \mathcal{D}(le.x) \wedge \mathcal{D}(y) \wedge y_t \preceq \mathcal{U}(atts, sc, le_t)(x)$$

The well-definedness of method calls $le.m(x)$ using **res** and **valres** specify that the frame of le and x must be disjoint. In this hybrid theory instead of using *sdisjoint*, as defined in ROOL to verify if two address do not overlap, the frame of a variable (address) can be easily calculated as

$$frame(x) \mathrel{\widehat{=}} share_S(x) \cup ext_S(x)$$

thus, the predicate $sdisjoint(le, x)$ can be replaced by

$$frame(le) \cap frame(x) = \emptyset$$

in the well-definedness rules for method calls, which is simpler to calculate.

Assignments

The five forms of assignments presented in the theory for pointers are reduced to three. They perform updates of *dts*, and also are presented as designs where preconditions include well-definedness restrictions. The definitions were changed to associate the addresses with their corresponding dynamic types. This information is crucial in type testing and casting, and to recover the value of an object, as previously discussed.

Different from variable (un)declaration, the definitions of assignments do not refer to those variables that remain unchanged w; this is due to the impact of changing the information in the pointer machine that can affect variables other than x which share values or locations.

The first form is assignment of values; we have joined definitions. The assignment $x := e$ where e is a constant is trivial, the second case was an assignment to a non-terminal node, in this integrated theory the only value assignable to such addresses is **null**. For both cases the definition below, applies.

$$x := e \mathrel{\widehat{=}} \mathbf{IT}\left(\mathcal{D}(x := e) \vdash \begin{pmatrix} A' = A \setminus ext_S(x) \land \\ V' = ((ext_S(x) \cup share_S(x)) \lhd V) \oplus \{a : (share_S(x) \setminus ext_S(x)) \bullet a \mapsto e_v\} \land \\ S' = ext_S(x) \lhd S \rhd ext_S(x) \land \\ dts' = ((ext_S(x) \cup share_S(x)) \lhd dts) \oplus \{a : (share_S(x) \setminus ext_S(x)) \bullet a \mapsto e_t\} \end{pmatrix}\right)$$

Notice that in the pointer theory $:-_i$ is a generalization of $:-_t$; we combined them into a single definition and introduced updates for *dts*.

The assignment of a newly created structure is different and defined in the sequel. In this case we have a structure associated to a type, and we record its information into the pointer machine with dynamic type information associated.

$$x :=_r \mathbf{new}\ N \mathrel{\widehat{=}} \mathbf{IT}\left(\mathcal{D}(x := \mathbf{new}\ N) \vdash \begin{pmatrix} A' = (A \setminus x\infty) \cup \{n : name \mid n \in \mathrm{dom}\, map \bullet x.n\} \land \\ V' = ((x\infty \cup \{x\}) \lhd V) \oplus \{n : name \mid n \in \mathrm{dom}\, map \bullet x.n \mapsto map(n)\} \land \\ S' = (x\infty \cup \{x\}) \lhd S \rhd (x\infty \cup \{x\}) \land \\ dts' = ((x\infty \cup \{x\}) \lhd dts) \oplus (\{x \mapsto N\} \cup \{x.f \mapsto aclos(f) \mid f \in \mathrm{dom}\, map\}) \land \\ w' = w \end{pmatrix}\right)$$

where $(N, map) = \mathbf{new}\ N$
with $aclos = \mathcal{U}(atts, cls, N)$
and $w \in in\alpha(x :=_r \mathbf{new}\ N) \setminus \{A, V, S, dts\}$

For pointer assignments the behaviour is the same as of that for pointers, but for each new shared entry a corresponding dynamic type is added to *dts*. Moreover, since '$:-_t$' is a generalization of '$:-_i$' only the general form is considered.

$$x :- y \mathrel{\widehat{=}} \mathbf{IT}\left(\begin{array}{l}\mathcal{D}(x :- e)\\ \vdash\\ \left(\begin{array}{l}A' = (A \setminus x{\infty}) \cup \{a : Ad \mid y.a \in A \bullet x.a\} \wedge\\ V' = (x{\infty} \lhd V) \oplus \{a : \mathrm{dom}\ V;\ fa : seq\ Label \mid a = y.fa \bullet x.fa \mapsto V(y.fa)\} \wedge\\ S' = (((x{\infty} \cup \{x\}) \lhd S \rhd (x{\infty} \cup \{x\})) \cup \{x \mapsto y\} \cup \{a : Ad \mid y.a \in A \bullet x.a \mapsto y.a\})^* \wedge\\ dts' = (x{\infty} \lhd dts) \oplus (\{a : \mathrm{dom}\ V;\ fa : seq\ Label \mid a = y.fa \bullet x.fa \mapsto dts(y.fa)\} \cup \{x \mapsto dts(y)\})\end{array}\right)\end{array}\right)$$

Method Call

The semantics of method calls has also to consider HCs from pointers and **PO**s; thus the new definition is

$$le.m(args) \mathrel{\widehat{=}} \mathbf{IT}(\{\mathcal{D}(le.m(args))\}_\perp;\ (pds_e \bullet p)(le, args))$$

$$\textbf{where}\ m = \mathtt{pds}_e \bullet \mathtt{p}$$

as explained before, but with the restrictions over observational variables of both theories and *dts*.

Call-by-reference With assignment of pointers, we can define another type of parameter passing mechanism: **ref**, passed by reference. The corresponding λ-abstraction is defined as follows.

$$(\mathbf{ref}\ v : T \bullet p) \mathrel{\widehat{=}} (\lambda\, w : \mathcal{N} \bullet (\mathbf{var}\ v : T;\ v :- w;\ p;\ \mathbf{end}\ v : T))$$

Its well-definedness rule is similar to the call-by-value.

Call-by-reference
$\mathcal{D}(le.m(e)) \ \mathrel{\widehat{=}}\ \mathcal{D}(le) \wedge le_v \neq \mathbf{null} \wedge compatible(le, m) \wedge e_t \preceq T$

$$\textbf{provided}\ \exists \mathtt{p} \bullet m = (\mathtt{ref\ x{:}T} \bullet \mathtt{p})$$

Example 8 (Integrated theory)**.** This is an example where two *Account*s share the same contact information. The program considered is presented in Figure 5.2. In its stepwise execution we consider the initial variables of object-orientation as described in Example 4, all pointer machine variables A, V and S are empty, and the dynamic type mapping *dts* is empty. The execution is concerned only with the states of *dts*, A, V and S since *cls*, *sc* and *atts* are not affected by the program execution. To reduce the example size we use the abbreviations *nu*, *ba*, *co*, *bo* and *ph* to refer to attributes *number*, *balance*, *contact*, *bonus* and *phone*, respectively, and for simplicity we use 0 for integers or **null** for objects as initial values of attributes, in spite of their non-determinism on initialization.

{ just before starting all variables are empty }
$A = \{\} \wedge dts = \{\} \wedge V = \{\} \wedge S = \{\}$

$$
\begin{aligned}
l_1&\text{: } \textbf{var } y : \mathit{Account};\\
l_2&\text{: } \textbf{var } x : \mathit{Account};\\
l_3&\text{: } y := \textbf{new } \mathit{BAccount};\\
l_4&\text{: } x :\!- y;\\
l_5&\text{: } x := \textbf{new } \mathit{Account};\\
l_6&\text{: } y.\mathit{contact} := \textbf{new } \mathit{Contact};\\
l_7&\text{: } x.\mathit{contact} :\!- y.\mathit{contact};\\
l_8&\text{: } x.\mathit{contact.phone} := 9;\\
l_9&\text{: } \textbf{end } y : \mathit{Account};\\
l_{10}&\text{: } \textbf{end } x : \mathit{Account}
\end{aligned}
$$

Figure 5.2: An object-oriented program with sharing.

l_1, l_2 :‘**var** $y : \mathit{Account}$; **var** $x : \mathit{Account}$’ { variable declaration }

$$A' = \begin{Bmatrix} y, \\ x \end{Bmatrix} \wedge \mathit{dts}' = \begin{Bmatrix} y \mapsto \mathit{Account}, \\ x \mapsto \mathit{Account} \end{Bmatrix} \wedge V' = \begin{Bmatrix} y \mapsto \textbf{null}, \\ x \mapsto \textbf{null} \end{Bmatrix} \wedge S' = \begin{Bmatrix} y \mapsto y, \\ x \mapsto x \end{Bmatrix}$$

l_3 :‘$y := \textbf{new } \mathit{BAccount}$’ { object creation and assignment ($:=_r$) for variable y }

$$A' = \begin{Bmatrix} x, \\ y, \\ y.nu, y.ba, \\ y.co, \\ y.bo \end{Bmatrix} \wedge \mathit{dts}' = \begin{Bmatrix} x \mapsto \mathit{Account}, \\ y \mapsto \mathit{BAccount}, \\ y.nu \mapsto \mathbb{Z}, y.ba \mapsto \mathbb{Z}, \\ y.co \mapsto \mathit{Contact}, \\ y.bo \mapsto \mathbb{Z} \end{Bmatrix} \wedge V' = \begin{Bmatrix} x \mapsto \textbf{null}, \\ y.nu \mapsto 0, y.ba \mapsto 0, \\ y.co \mapsto \textbf{null}, \\ y.bo \mapsto 0 \end{Bmatrix} \wedge S' = \begin{Bmatrix} y \mapsto y, \\ x \mapsto x \end{Bmatrix}$$

l_4 :‘$x :\!- y$’ { pointer assignment, associate a chain of references }

$$A' = \begin{Bmatrix} x, \\ x.nu, x.ba, \\ x.co, \\ x.bo, \\ y, \\ y.nu, y.ba, \\ y.co, \\ y.bo \end{Bmatrix} \wedge \mathit{dts}' = \begin{Bmatrix} x \mapsto \mathit{BAccount}, \\ x.nu \mapsto \mathbb{Z}, x.ba \mapsto \mathbb{Z}, \\ x.co \mapsto \mathit{Contact}, \\ x.bo \mapsto \mathbb{Z}, \\ y \mapsto \mathit{BAccount}, \\ y.nu \mapsto \mathbb{Z}, y.ba \mapsto \mathbb{Z}, \\ y.co \mapsto \mathit{Contact}, \\ y.bo \mapsto \mathbb{Z} \end{Bmatrix} \wedge V' = \begin{Bmatrix} x.nu \mapsto 0, \\ x.ba \mapsto 0, \\ x.co \mapsto \textbf{null} \\ x.bo \mapsto 0 \\ y.nu \mapsto 0, \\ y.ba \mapsto 0, \\ y.co \mapsto \textbf{null}, \\ y.bo \mapsto 0 \end{Bmatrix} \wedge S' = \begin{Bmatrix} x \mapsto y, \\ x \mapsto x, \\ x.nu \mapsto y.nu, \\ x.ba \mapsto y.ba, \\ x.co \mapsto y.co, \\ x.bo \mapsto y.bo \end{Bmatrix}^*$$

l_5 :'$x :=$ **new** *Account*' { object creation and assignment ($:=_r$) for variable x }

$$A' = \left\{\begin{array}{l} x, \\ x.nu, x.ba, \\ x.co, \\ y, \\ y.nu, y.ba, \\ y.co, \\ y.bo \end{array}\right\} \wedge dts' = \left\{\begin{array}{l} x \mapsto Account, \\ x.nu \mapsto \mathbb{Z}, x.ba \mapsto \mathbb{Z}, \\ x.co \mapsto Contact, \\ y \mapsto BAccount, \\ y.nu \mapsto \mathbb{Z}, y.ba \mapsto \mathbb{Z}, \\ y.co \mapsto Contact, \\ y.bo \mapsto \mathbb{Z} \end{array}\right\} \wedge V' = \left\{\begin{array}{l} x.nu \mapsto 0, x.ba \mapsto 0, \\ x.co \mapsto \textbf{null} \\ y.nu \mapsto 0, y.ba \mapsto 0, \\ y.co \mapsto \textbf{null}, \\ y.bo \mapsto 0 \end{array}\right\} \wedge S' = \left\{\begin{array}{l} x \mapsto y, \\ x \mapsto x \end{array}\right\}$$

l_6 :'$y.contact :=$ **new** *Contact*'{ object creation and assignment ($:=_r$) to attribute *contact* of x }

$$A' = \left\{\begin{array}{l} x, \\ x.nu, x.ba, \\ x.co, \\ y, \\ y.nu, y.ba, \\ y.co, \\ y.bo, \\ y.co.ph \end{array}\right\} \wedge dts' = \left\{\begin{array}{l} x \mapsto Account, \\ x.nu \mapsto \mathbb{Z}, x.ba \mapsto \mathbb{Z}, \\ x.co \mapsto Contact, \\ y \mapsto BAccount, \\ y.nu \mapsto \mathbb{Z}, y.ba \mapsto \mathbb{Z}, \\ y.co \mapsto Contact, \\ y.co.ph \mapsto \mathbb{Z}, \\ y.bo \mapsto \mathbb{Z} \end{array}\right\} \wedge V' = \left\{\begin{array}{l} x.nu \mapsto 0, x.ba \mapsto 0, \\ x.co \mapsto \textbf{null} \\ y.nu \mapsto 0, y.ba \mapsto 0, \\ y.co.ph \mapsto 0, \\ y.bo \mapsto 0 \end{array}\right\} \wedge S' = \left\{\begin{array}{l} x \mapsto y, \\ x \mapsto x \end{array}\right\}$$

l_7 :'$x.contact :- y.contact$' { pointer assignment }

$$A' = \left\{\begin{array}{l} x, \\ x.nu, x.ba, \\ x.co, \\ x.co.ph, \\ y, \\ y.nu, y.ba, \\ y.co, \\ y.bo, \\ y.co.ph \end{array}\right\} \wedge dts' = \left\{\begin{array}{l} x \mapsto Account, \\ x.nu \mapsto \mathbb{Z}, x.ba \mapsto \mathbb{Z}, \\ x.co \mapsto Contact, \\ x.co.ph \mapsto \mathbb{Z}, \\ y \mapsto BAccount, \\ y.nu \mapsto \mathbb{Z}, y.ba \mapsto \mathbb{Z}, \\ y.co \mapsto Contact, \\ y.co.ph \mapsto \mathbb{Z}, \\ y.bo \mapsto \mathbb{Z} \end{array}\right\} \wedge V' = \left\{\begin{array}{l} x.nu \mapsto 0, \\ x.ba \mapsto 0, \\ x.co.ph \mapsto 0 \\ y.nu \mapsto 0, \\ y.ba \mapsto 0, \\ y.co.ph \mapsto 0, \\ y.bo \mapsto 0 \end{array}\right\} \wedge S' = \left\{\begin{array}{l} x \mapsto y, \\ x \mapsto x, \\ x.co \mapsto y.co, \\ x.co.ph \mapsto y.co.ph \end{array}\right\}^*$$

l_8 :'$x.contact.phone := 9$' { value assignment to a shared attribute, 9 stands for ($\mathbb{Z}$,9) }

$$A' = \left\{\begin{array}{l} x, \\ x.nu, x.ba, \\ x.co, \\ x.co.ph, \\ y, \\ y.nu, y.ba, \\ y.co, \\ y.bo, \\ y.co.ph \end{array}\right\} \wedge dts' = \left\{\begin{array}{l} x \mapsto Account, \\ x.nu \mapsto \mathbb{Z}, x.ba \mapsto \mathbb{Z}, \\ x.co \mapsto Contact, \\ x.co.ph \mapsto \mathbb{Z}, \\ y \mapsto BAccount, \\ y.nu \mapsto \mathbb{Z}, y.ba \mapsto \mathbb{Z}, \\ y.co \mapsto Contact, \\ y.co.ph \mapsto \mathbb{Z}, \\ y.bo \mapsto \mathbb{Z} \end{array}\right\} \wedge V' = \left\{\begin{array}{l} x.nu \mapsto 0, \\ x.ba \mapsto 0, \\ x.co.ph \mapsto 9 \\ y.nu \mapsto 0, \\ y.ba \mapsto 0, \\ y.co.ph \mapsto 9, \\ y.bo \mapsto 0 \end{array}\right\} \wedge S' = \left\{\begin{array}{l} x \mapsto y, \\ x \mapsto x, \\ x.co \mapsto y.co, \\ x.co.ph \mapsto y.co.ph \end{array}\right\}^*$$

l_9 :‘**end** $y : Account$’{ undeclaration }

$$A' = \begin{Bmatrix} x, \\ x.nu, \\ x.ba, \\ x.co, \\ x.co.ph \end{Bmatrix} \land dts' = \begin{Bmatrix} x \mapsto Account, \\ x.nu \mapsto \mathbb{Z}, \\ x.ba \mapsto \mathbb{Z}, \\ x.co \mapsto Contact, \\ x.co.ph \mapsto \mathbb{Z} \end{Bmatrix} \land V' = \begin{Bmatrix} x.nu \mapsto 0, \\ x.ba \mapsto 0, \\ x.co.ph \mapsto 9 \end{Bmatrix} \land S' = \left\{ x \mapsto x \right\}$$

l_{10} :‘**end** $x : Account$’{ undeclaration }

$A' = \{\} \land dts' = \{\} \land V' = \{\} \land S' = \{\}$

This example shows that the integration is feasible. Other lengthier examples can be derived using sharing and method calls. □

5.4 Conclusions

This chapter has shown how the theory for object-orientation is progressively adapted to consider a pointer machine introducing type information to records. The theory proposed by Harwood *et al.* has been presented with a simple example to show how the constructs of this theory work, then the representation of a value for the object-orientation theory was changed to handle the pointer machine. Many concepts are common for both theories but, the UTP framework allowed us to combine concepts in single definitions. This task would become easier if one theory were subset of the other, or related by a Galois connection as presented in Section 3.7.

We have seen that the integration process required some adjustments in some concepts such as the interpretation of values, the value of a variable (address) is a combined version of the concept of value for object-orientation (typed and hierarchically structured) and the concept of value for pointers (untyped and flat). As a consequence a new observational variable *dts* with its associated healthiness conditions **PO** was introduced to record dynamic types of addresses. Moreover, we have seen that if healthiness conditions are too restrictive the integration process may become impossible; in the case of **HP3** we had to generalize its definition to allow further integration. This made clear that the healthiness conditions of a theory should only refer to the observational variables of that theory, and might not use the alphabet as reference for restrictions, as in the original **HP3** definition.

Other important observation is that all elements of both theories, combined or not, must be **OO**, **HP** and **PO** healthy; for example, in class, attribute or method declarations, the **OO** application in object-orientation with copy semantics should be replaced by this composition in the object-orientation with reference semantics. In Appendix D all definitions with all required adjustments are presented.

An important result of the integration process is its preservation of many definitions for object-orientation features such as subtyping, recursion and dynamic binding which did not change. The introduction of a memory model impacted the process of object creation and assignment commands, which had to be redefined with respect to both theories. The methods definitions and calls, however, present very similar behaviour to the versions in Chapter 4. In the case of method declarations, however, for the integrated theory the $\bot_{oo}$ used in method texts must be replaced by $\bot_{po}$ which is defined as $\bot_{po} \mathrel{\widehat{=}} \mathbf{OO} \circ \mathbf{HP} \circ \mathbf{PO}(\bot)$.

The definition of assignments of values in the pointer theory seems to lack a definition of this kind $x := y$, that is, an assignment which copies the value of y (address) to x, like in the copy semantics. Its definition is similar to that of ' :– ', but the inclusion of pairs in S is not considered, therefore, x and y are not expected to share the same location. After this assignment they have the same value, but are not aliased. The definition of such assignment is

$$
\begin{array}{l}
x := y \mathrel{\widehat{=}} \\
\mathbf{IT}\left(\begin{array}{l}
\mathcal{D}(x := y) \\
\vdash \\
\left(\begin{array}{l}
A' = (A \setminus x\infty) \cup \{a : Ad \mid y.a \in A \bullet x.a\} \land \\
dts' = (x\infty \lhd dts) \oplus (\{a : \mathrm{dom}\ V;\ fa : seq\ Label \mid a = y.fa \bullet x.fa \mapsto dts(y.fa)\} \cup \{x \mapsto dts(y)\}) \land \\
V' = (x\infty \lhd V) \oplus \{a : \mathrm{dom}\ V;\ fa : seq\ Label \mid a = y.fa \bullet x.fa \mapsto V(y.fa)\} \land \\
S' = ((x\infty \cup \{x\}) \lhd S \rhd (x\infty \cup \{x\}))
\end{array}\right)
\end{array}\right)
\end{array}
$$

Since an expression can be a variable (address), these definitions are required to complete the formalisation of assignments. They are also required for method calls; for example, if we have

$$\ldots;\ \mathbf{var}\ y : Account;\ y := \mathbf{new}\ BAccount;\ le.m(y);\ \ldots$$

where m is defined for the type of le and has a '**val** x : *Account*' parameter, the copy of y to x is required. The assignments of expressions different of variables in pointer assignments are not allowed, therefore, the form $x :\!- e$ does not exist.

The benefits of all these kinds of assignments is that depending on the kind of assignments we use in the program, or in the λ-abstraction interpretations, we should have a mix of pointer and copy semantics as in Java, where primitives are passed by value and objects are passed by references. For example, suppose we have only one kind of parameter named **par**, to provide a Java like semantics for method calls then we could have the following λ-abstractions

$$
\begin{array}{l}
(\mathbf{par}\ v : \mathbb{B} \bullet p) \mathrel{\widehat{=}} (\lambda\, w : \mathcal{N} \bullet (\mathbf{var}\ v : T;\ v := w;\ p;\ \mathbf{end}\ v : T)) \\
(\mathbf{par}\ v : \mathbb{Z} \bullet p) \mathrel{\widehat{=}} (\lambda\, w : \mathcal{N} \bullet (\mathbf{var}\ v : T;\ v := w;\ p;\ \mathbf{end}\ v : T)) \\
(\mathbf{par}\ v : T \bullet p) \mathrel{\widehat{=}} (\lambda\, w : \mathcal{N} \bullet (\mathbf{var}\ v : T;\ v :\!- w;\ p;\ \mathbf{end}\ v : T))
\end{array}
$$

where primitive values are copied to new local variables, and objects are aliased. In this case, the

interpretation of the dynamic binding of calling $a.credit(10)$ in Example 6 would be

var self : ***Object***;
  self :– a;
  var $x : \mathbb{Z}$;
    $x := 10$;
    self.$bonus$:= **self**.$bonus + 1$;
    self.$balance$:= **self**.$balance + x$;
  end $x : \mathbb{Z}$;
end self : ***Object***

This Chapter has shown that the integration of theories in the UTP can be performed and the result reuse concepts of both theories considered. Depending on the concepts of the theories under consideration the integration can become easier if links between theories are found. The integration of theories we have presented follows a compositional approach from a simpler theory to a more complex one formed by concepts of different theories. The composition process, however, was not as simple as wanted.

Chapter 6

Laws for Object-Orientation

This chapter presents laws for object-oriented programs. We start, in Section 6.1, with a short introduction to the topic and discuss related works. In Section 6.2 we present the laws themselves and discuss their applicability. Finally, in Section 6.3 we conclude our discussion on object-oriented laws.

6.1 Introduction

An important aspect of object-oriented programming is the use of program transformations during software construction or maintenance. In spite of their informal descriptions, some transformation techniques have been introduced for different programming languages, like Java and C++, and their corresponding IDEs. In his classic work [FBB+99], Fowler catalogs a set of program transformations (refactorings), and describes how to apply them. The correctness of the refactorings, however, is only informally addressed based on test cases executed before and after the program manipulation.

Following an algebraic approach, in [BS00] we find a set of laws of object-orientation for a subset of sequential Java enriched with refinement constructs. In [Cor04], Cornélio describes a subset of Fowler's refactorings in the same language of [BS00] and proves that these transformations are sound according to a semantics defined in terms of weakest preconditions. This set of refactorings, however, are restricted to those which not required a reference semantics.

In this chapter we present some laws of object-oriented programs, which are inspired by the works reported in [BS00, CN00, BSCC04, Cor04], providing laws applicable to open (libraries of classes) or closed systems; this taxonomy is characterised in the next subsection. Our sources of inspiration, however, deal only with closed programs, in this way we can say that we have a more general approach. Moreover, we show that out theory for object-orientation can be used to prove the soundness of these laws in a relatively concise way. This is a major contribution of this work.

Closed Vs Open Systems

The objective of this subsection is to contextualise two kinds of object-oriented systems that restrict the set of laws of programming that can be used. More specifically, we provide an short overview of why closed systems are of simpler refactoring than open systems.

We say that a system is closed if their component parts are fully described and composed before its analysis or execution. In this context, we have a sequence of object-oriented declarations which are expected to work with only one specific program, that is, a dedicated library which can be used only by one program. We could characterise such systems as

$$cds \bullet c$$

where *cds* stands for declarations and c is the program for which these declarations are expected to provide services. In this kind of systems one can identify what services of *cds* are used by c and inside *cds* what features of one class are referred by the others, and, moreover, other programs which would rely on services provided by *cds* are not considered.

With all these detailed information about the declarations, and the program itself, this kind of program becomes a good candidate for refactoring. These systems are those considered by [BS00, CN00, BSCC04, Cor04]. A law for *class removal*, shortly described below, when applied from left to right ('$\rightarrow$'), for example, can be described as: let N be a class, if in *cds* there aren't references to N or to its attributes or methods besides their own declarations, and in c N is not used, then

$$\boxed{\textbf{class}\ N;\ cds \bullet c} = \boxed{cds \bullet c}$$

Observe also that the opposite direction (from right to left – '$\leftarrow$') can be interpreted as a law for *class introduction*. In this case, we can introduce a class which has not been previously defined. Since the laws is applicable in both directions ('$\leftrightarrows$') we use the '=' symbol. This is just one example of the many laws provided for closed systems.

Now, imagine that you don't have an specify program c to consider; it is left unspecified and can use any services provided by *cds* or even extend *cds* and them perform something, this is exactly the case of an open system. An open systems is characterised as a system where its components are unknown until its use. That is, we have a sequence of declarations, but we don't know how many programs are using or extending such declarations, and who they are. A general form would be

$$cds;\ \ldots \bullet c;\ \ldots$$

where the set of classes and programs that uses the declarations are undefined, probably infinite. It poses one important general restriction if we are expected to perform refactorings in *cds*; since

we don't know which services of *cds* are being used we cannot perform refactorings that reduce the set of such services, i.e. remove a class as in the case of closed systems.

Notice, however, that a class cannot be remove, but it can be introduced without any restriction, besides the fact that it cannot be already defined. That is, for open systems the law is valid only in one direction (from right to left).

$$\boxed{\textbf{class}\ N;\ cds;\ \ldots \bullet c;\ \ldots} \leftarrow \boxed{cds;\ \ldots \bullet c;\ \ldots}$$

The left side contains a larger set of classes and thus can be used to constructs a larger set of programs than the right side of the law. In the following section we provide the description and proofs of laws considering these two contexts.

6.2 Laws

Most part of the basic laws of programming for sequential programs were already stated and proved in [HH98]. For example: (i) '(**var** $x : T;\ c) = c$', if x is not free in c; or (ii) '$(x := x) = I\!I$'. This section shows some laws, not an extensive list, valid both for open or closed systems. We concentrate on laws for object-oriented expressions, methods and attributes, but different of healthiness conditions the proofs are not presented in the appendix.

The following two laws exemplify that some proofs in the UTP can be very simple, depending on how the constructs of object-orientation are modeled. In a context where B is a subclass of A, a disjunction of type tests over these two types can be reduced to a type test on A. Informally, if a given expression e is an instance of type B, this instance is of type A too, which is a more general condition. This is formally described and proved below.

Law 2. *<∨-subtyping>*

$$\boxed{e\ \textbf{is}\ B \vee e\ \textbf{is}\ A} \quad = \quad \boxed{e\ \textbf{is}\ A}$$

provided

$$B \preceq A.$$

Proof.

$$\begin{array}{ll}
& LHS \\
= & (e_t \preceq B \vee e_t \preceq A) \wedge B \preceq A & \textit{[semantics of type test, assumption]} \\
= & (e_t \preceq B \wedge B \preceq A) \vee (e_t \preceq A \wedge B \preceq A) & \textit{[∧-distribution]} \\
= & (e_t \preceq A \wedge B \preceq A) \vee (e_t \preceq A \wedge B \preceq A) & \textit{[⪯-transitivity]} \\
= & e_t \preceq A & \textit{[assumption]} \\
= & e\ \textbf{is}\ A & \textit{[type test]} \\
= & RHS &
\end{array}$$

□

On the other hand, if we have that B is a subclass of A in a conjunction of type tests, we can reduce the condition to e **is** B.

Law 3. *<∧-subtyping>*

$$\boxed{e \textbf{ is } B \wedge e \textbf{ is } A} \quad = \quad \boxed{e \textbf{ is } B}$$

provided

$$B \preceq A.$$

Proof.

$$\begin{array}{ll} RHS & \textit{[semantics of type test, assumption]} \\ = e_t \preceq B \wedge B \preceq A & \textit{[}\preceq\textit{-transitivity]} \\ = e_t \preceq B \wedge e_t \preceq A \wedge B \preceq A & \textit{[type test]} \\ = (e \textbf{ is } B \wedge e \textbf{ is } A) \wedge B \preceq A & \textit{[assumption]} \\ = LHS & \square \end{array}$$

Notice that these two laws, and the forthcoming, are valid in a reference semantics context as well. This is because the value of an expression is formed by a pair (e_t, e_v) in both theories for object-orientation, either with reference semantics or not, as presented in Chapters 4 and 5.

A more interesting law is that the introduction of a method redefinition (m) with the same body (ma) of the superclass (A) into a given subclass (B) does not change the expected result of the method call for any instances of classes A and B. That is, the command executed will always be ma, or for invalid method calls it will be abort. The HCs required for law proofs are those **OO** which are part of **IT**, and the $\bot$ used in methods can be easily replaced by $\bot_d$, $\bot_{oo}$ and $\bot_{po}$ for an specific context.

Law 4. *<introduce trivial method redefinition>*

$$\boxed{\textbf{meth } A\ m = (pds \bullet ma)} \quad = \quad \boxed{\begin{array}{l} \textbf{meth } A\ m = (pds \bullet ma)\,; \\ \textbf{meth } B\ m = (pds \bullet ma) \end{array}}$$

provided

$$B \prec A \wedge m \notin in\alpha.$$

Proof.

$$\begin{array}{l} \textbf{meth } A\ m = (pds \bullet ma)\,;\ \textbf{meth } B\ m = (pds \bullet ma) \\ \qquad\qquad \textit{[definition and } \mathtt{pds}_e = \mathtt{valres\ self{:}Object;\ pds}\textit{]} \\ = \textbf{IT}(\textbf{var } m;\ (A \in cls \vdash m' = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot))); \\ \quad \textbf{IT}\,(B \in cls \wedge \exists\ \mathtt{q} \bullet m = \mathtt{pds}_e \bullet \mathtt{q} \vdash\ (\exists\, \mathtt{q} \bullet (m = \mathtt{pds}_e \bullet \mathtt{q} \wedge m' = \mathtt{pds}_e \bullet join(B, \mathtt{ma}, \mathtt{q})))) \\ \qquad\qquad \textit{[assuming } \textbf{IT} \textit{ and premisses } (B \prec A \Rightarrow A \in cls \wedge B \in cls)\textit{]} \\ = \textbf{var } m;\ \textbf{true} \vdash m' = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot); \end{array}$$

$$\exists\, \mathtt{q} \bullet (m = \mathtt{pds}_e \bullet \mathtt{q}) \vdash \exists\, \mathtt{q} \bullet \big(m = \mathtt{pds}_e \bullet \mathtt{q} \wedge m' = \mathtt{pds}_e \bullet \mathit{join}(B, \mathtt{ma}, \mathtt{q})\big)$$

[variable declaration, ;-associativity]

$$= \exists\, m \bullet \begin{pmatrix} \mathbf{true} \vdash m' = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot); \\ \exists\, \mathtt{q} \bullet (m = \mathtt{pds}_e \bullet \mathtt{q}) \vdash \exists\, \mathtt{q} \bullet \big(m = \mathtt{pds}_e \bullet \mathtt{q} \wedge m' = \mathtt{pds}_e \bullet \mathit{join}(B, \mathtt{ma}, \mathtt{q})\big) \end{pmatrix}$$

*['**T3**' of designs]*

$$= \exists\, m \bullet \begin{pmatrix} \mathbf{true} \wedge \neg(m' = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot);\ \neg\, \exists\, \mathtt{q} \bullet (m = \mathtt{pds}_e \bullet \mathtt{q}))\text{(i)} \\ \vdash \\ m' = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot);\ \exists\, \mathtt{q} \bullet \big(m = \mathtt{pds}_e \bullet \mathtt{q} \wedge m' = \mathtt{pds}_e \bullet \mathit{join}(B, \mathtt{ma}, \mathtt{q})\big)\ \text{(ii)} \end{pmatrix} \text{(0)}$$

(i) [by sequential composition]

$= \mathbf{true} \wedge \neg \big(\exists\, m_0 \bullet m_0 = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot) \wedge \neg\, \exists\, \mathtt{q} \bullet (m_0 \neq \mathtt{pds}_e \bullet \mathtt{q})\big)$

[propositional calculus]

$= \mathbf{true} \wedge \neg \big(\exists\, m_0, \neg\, \exists\, \mathtt{q} \bullet m_0 = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot) \wedge m_0 = \mathtt{pds}_e \bullet \mathtt{q}\big)$

[contradiction]

$= \mathbf{true} \wedge \neg \big(\mathbf{false}\big)$

[propositional calculus]

$= \mathbf{true}$

(ii) [by sequential composition]

$= \exists\, m_0 \bullet m_0 = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot) \wedge \exists\, \mathtt{q} \bullet \big(m_0 = \mathtt{pds}_e \bullet \mathtt{q} \wedge m' = \mathtt{pds}_e \bullet \mathit{join}(B, \mathtt{ma}, \mathtt{q})\big)$

[propositional calculus]

$= \exists\, m_0, \mathtt{q} \bullet m_0 = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot) \wedge m_0 = \mathtt{pds}_e \bullet \mathtt{q} \wedge m' = \mathtt{pds}_e \bullet \mathit{join}(B, \mathtt{ma}, \mathtt{q})$

[∃-elimination]

$= m' = \mathtt{pds}_e \bullet \mathit{join}(B, \mathtt{ma}, \mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot)$

[join, $B \prec A$]

$= m' = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ B} \rhd (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot))$

(0)

$= \exists\, m \bullet \mathbf{true} \vdash m' = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ B} \rhd (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot))$ *[by **L7** of conditional]*

$= \exists\, m \bullet \mathbf{true} \vdash m' = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ B} \vee \mathtt{self\ is\ A} \rhd \bot)$ *[by **Law 2**]*

$= \exists\, m \bullet \mathbf{true} \vdash m' = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot)$ *[variable]*

$= \mathbf{var}\ m;\ \mathbf{true} \vdash m' = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot)$ *[assuming **IT** and premisses $(A \prec B)$]*

$= \mathbf{IT}(\mathbf{var}\ m;\ \big(A \in \mathit{cls} \vdash m' = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot)\big))$ *[method definition]*

$= \mathbf{meth}\ A\ m = \big(\mathit{pds} \bullet \mathit{ma}\big)$ □

Also, we can combine method redefinitions, as long as we preserve the behaviour of that method for the handled subclasses.

Law 5. *<move redefined method to superclass>*

$$\boxed{\begin{array}{l} \mathbf{meth}\ A\ m = \big(\mathit{pds} \bullet \mathit{ma}\big)\,; \\ \mathbf{meth}\ B\ m = \big(\mathit{pds} \bullet \mathit{mb}\big) \end{array}} \quad = \quad \boxed{\mathbf{meth}\ A\ m = \big(\mathit{pds} \bullet \mathit{mb} \lhd \mathbf{self\ is}\ B \rhd \mathit{ma}\big)}$$

provided

$$B \prec A \wedge m \notin \mathit{in}\alpha.$$

Proof.

$\mathbf{meth}\ A\ m = \big(\mathit{pds} \bullet \mathit{ma}\big)\,;\ \mathbf{meth}\ B\ m = \big(\mathit{pds} \bullet \mathit{mb}\big)$

[definition and $\mathtt{pds}_e = \mathtt{valres\ self{:}Object;\ pds}$*]*

$= \mathbf{IT}(\mathbf{var}\ m;\ \big(A \in cls \vdash m' = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot)\big));$
$\quad \mathbf{IT}\ \big(B \in cls \wedge \exists\ \mathtt{q} \bullet m = \mathtt{pds}_e \bullet \mathtt{q} \vdash\ \big(\exists\, \mathtt{q} \bullet \big(m = \mathtt{pds}_e \bullet \mathtt{q} \wedge m' = \mathtt{pds}_e \bullet \mathit{join}(B, \mathtt{mb}, \mathtt{q})\big)\big)\big)$

[assuming **IT** *and premisses* $(B \prec A \Rightarrow A \in cls \wedge B \in cls)$*]*

$= \mathbf{var}\ m;\ \mathbf{true} \vdash m' = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot);$
$\quad \exists\, \mathtt{q} \bullet (m = \mathtt{pds}_e \bullet \mathtt{q}) \vdash \exists\, \mathtt{q} \bullet \big(m = \mathtt{pds}_e \bullet \mathtt{q} \wedge m' = \mathtt{pds}_e \bullet \mathit{join}(B, \mathtt{mb}, \mathtt{q})\big)$

[variable declaration, ;-associativity]

$$= \exists\, m \bullet \begin{pmatrix} \mathbf{true} \vdash m' = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot); \\ \exists\, \mathtt{q} \bullet (m = \mathtt{pds}_e \bullet \mathtt{q}) \vdash \exists\, \mathtt{q} \bullet \big(m = \mathtt{pds}_e \bullet \mathtt{q} \wedge m' = \mathtt{pds}_e \bullet \mathit{join}(B, \mathtt{mb}, \mathtt{q})\big) \end{pmatrix}$$

*[***T3'** *of designs]*

$$= \exists\, m \bullet \begin{pmatrix} \mathbf{true} \wedge \neg(m' = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot);\ \neg\, \exists\, \mathtt{q} \bullet (m = \mathtt{pds}_e \bullet \mathtt{q}))\text{(i)} \\ \vdash \\ m' = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot);\ \exists\, \mathtt{q} \bullet \big(m = \mathtt{pds}_e \bullet \mathtt{q} \wedge m' = \mathtt{pds}_e \bullet \mathit{join}(B, \mathtt{mb}, \mathtt{q})\big)\ \text{(ii)} \end{pmatrix} (0)$$

(i) [by sequential composition]

$= \mathbf{true} \wedge \neg \big(\exists\, m_0 \bullet m_0 = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot) \wedge \neg\, \exists\, \mathtt{q} \bullet (m_0 \neq \mathtt{pds}_e \bullet \mathtt{q})\big)$

[propositional calculus]

$= \mathbf{true} \wedge \neg \big(\exists\, m_0, \neg\, \exists\, \mathtt{q} \bullet m_0 = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot) \wedge m_0 = \mathtt{pds}_e \bullet \mathtt{q}\big)$

[contradiction]

$= \mathbf{true} \wedge \neg \big(\mathbf{false}\big)$

[propositional calculus]

$= \mathbf{true}$

(ii) [by sequential composition]

$= \exists\, m_0 \bullet m_0 = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot) \wedge \exists\, \mathtt{q} \bullet \big(m_0 = \mathtt{pds}_e \bullet \mathtt{q} \wedge m' = \mathtt{pds}_e \bullet \mathit{join}(B, \mathtt{mb}, \mathtt{q})\big)$

[propositional calculus]

$= \exists\, m_0, \mathtt{q} \bullet m_0 = \mathtt{pds}_e \bullet (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot) \wedge m_0 = \mathtt{pds}_e \bullet \mathtt{q} \wedge m' = \mathtt{pds}_e \bullet \mathit{join}(B, \mathtt{mb}, \mathtt{q})$

[$\exists$*-elimination]*

$= m' = \mathtt{pds}_e \bullet \mathit{join}(B, \mathtt{mb}, \mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot)$

[join, $B \prec A$*]*

$= m' = \mathtt{pds}_e \bullet (\mathtt{mb} \lhd \mathtt{self\ is\ B} \rhd (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot))$

(0)

$= \exists\, m \bullet \mathbf{true} \vdash m' = \mathtt{pds}_e \bullet (\mathtt{mb} \lhd \mathtt{self\ is\ B} \rhd (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot))$

[by **Law 3***]*

$= \exists\, m \bullet \mathbf{true} \vdash m' = \mathtt{pds}_e \bullet (\mathtt{mb} \lhd \mathtt{self\ is\ B} \wedge \mathtt{self\ is\ A} \rhd (\mathtt{ma} \lhd \mathtt{self\ is\ A} \rhd \bot))$

[by **L3** *of conditional]*

$= \exists\, m \bullet \mathbf{true} \vdash m' = \mathtt{pds}_e \bullet ((\mathtt{mb} \lhd \mathtt{self\ is\ B} \rhd \mathtt{ma}) \lhd \mathtt{self\ is\ A} \rhd \bot)$

[variable]

$= \mathbf{var}\ m;\ \mathbf{true} \vdash m' = \mathtt{pds}_e \bullet ((\mathtt{mb} \lhd \mathtt{self\ is\ B} \rhd \mathtt{ma}) \lhd \mathtt{self\ is\ A} \rhd \bot)$

[assuming **IT** *and premisses* $(A \prec B)$*]*

$= \mathbf{var}\ m;\ \mathbf{IT}(A \in cls \vdash m' = \mathtt{pds}_e \bullet ((\mathtt{mb} \lhd \mathtt{self\ is\ B} \rhd \mathtt{ma}) \lhd \mathtt{self\ is\ A} \rhd \bot))$

[method definition]

$= \mathbf{meth}\ A\ m = \big(pds \bullet mb \lhd \mathbf{self\ is}\ B \rhd ma\big)$ □

These laws are also valid for copy or references semantics. Their validity relies on method variables (re)declarations which have the same format in both contexts.

6.3 Conclusions

As Hoare and He have done for other theories, we have introduced laws and proved them using our definitions. This initial set of laws provide evidence that we are capable of comparing implementations of libraries and conclude whether they are equivalent or not.

This set of laws tends to increase as we extend the approach to handle more closed programs refactorings, considering, for example, laws to allow introduction or removal of classes from a program that become valid in this context. For instance, if a given class is not used anywhere in the declarations or in the program main body, we can just drop its declaration as we have informally described in Section 6.1.

A deeper analysis of other laws presented in [BSCC04] will reveal other laws valid for open systems and those defined for closed programs are the target of our next research steps. These new laws may also require extra healthiness conditions to be identified and used to characterise a more restricted set of valid programs.

For example, for open systems we cannot remove classes from a class domain, but we can extend the set of classes; therefore a HC could be

$$\mathbf{OO14} \mathrel{\widehat{=}} P \land cls \subseteq cls'$$

We could also generalise the idea of moving an attribute up as a a healthiness condition

$$\mathbf{OO15} \mathrel{\widehat{=}} P \land \forall\, C : \mathrm{dom}\, atts \bullet \mathcal{U}(atts, sc, C) \subseteq \mathcal{U}(atts', sc', C)$$

Notice, however, that these conditions are trivially satisfied in languages without the possibility of undeclaration of classes and attributes, for closed or open programs.

For the correct characterisation of laws involving two different groups of classes we should have, rather, the test

$$cls_{LHS} \subseteq cls_{RHS}$$

where the left-hand side of the law represents the library before introducing a new element, captured by the right-hand side.

$$\forall\, C : \mathrm{dom}\, atts_{LHS} \bullet \mathcal{U}(atts_{LHS}, sc_{LHS}, C) \subseteq \mathcal{U}(atts_{RHS}, sc_{RHS}, C)$$

where the $atts_{LHS}$ and sc_{LHS} represent the mapping of attributes and the set of classes before moving the attribute up.

Chapter 7

Conclusions

In this Chapter we present a resume of the thesis and its results in Section 7.1. Section 7.3 presents the next planned steps to thesis extension and finalisation. Finally, in Section 7.4, we highlight some of possible extensions of our work.

7.1 Resume and Results

We started in Chapter 1 and Chapter 2 with an overview of our motivation, works in the area of formalisation of object-orientation, and resumed some of the most important representant. We have seen that many works were progressively developed and evolved to describe features of object-orientation such as subtyping and dynamic binding, specially after 90's.

The semantics frameworks of most part of them, however, are different and their approach are non-compositional. By non-compositional, we mean, different formalisms cannot be combined and give rise to more elaborated theories (descriptions) for representing richer paradigms in a simple way. To solve this problem, Hoare and He proposed a semantic framework named Unifying Theories of Programming (UTP) capable of describing many different paradigms using a common formalism and thus allow their straightforward comparison and combination. In Chapter 3 we provided an introductory overview of the UTP framework presenting their important concepts to understand the following chapters of the thesis.

Next, in Chapter 4 we have demonstrated that object-orientation with subtyping, data inheritance and dynamic binding can be defined in the UTP, using a theory that combines designs and higher-order procedures. In particular, we have introduced three observational variables to capture information about class declarations, extra variables xt and xt', for each programming variable x, to capture the type of the variables, and, finally, variables m and m' to capture the meaning (parameters and body) of each method named m. In our theory, recursion and mutual recursion (with minor restrictions) are handled in a simple way.

The concept of variable in the object-orientation context requires explicit typing information to allow the specification of well-definedness rules for expressions and commands, and to provide the correct semantics of object-oriented expressions and commands such as assignments, conditional, and method calls. We have a number of restrictions related to typing; all operations, and commands, over variables, values and expressions are checked. We have seen that invalid declarations and commands associated with object-oriented elements lead to abortion; in other words, the meaning of a badly-typed program is $\bot_{oo}$ (in the integrated theory $\bot_{po}$), which has the most unpredictable behaviour that we would expect.

In contrast to [HLL05], we do not use a runtime environment nor concentrate on a particular language; in the first moment we adopt a copy semantics, as in [CN00], and general concepts of object-orientation. We observe that this formalism is enough to write object-oriented programs but its applicability in real world language modeling can be restricted because of its copy semantics. The next natural step was to extend the formalism, in a compositional manner to handle object-oriented programs with references.

In this way, in Chapter 5 we presented the concept of sharing as a separate and complementary theory proposed by Harwood *et al.*. Then, we included extra information, healthiness conditions, and review well-definedness, expressions and commands for a hybrid theory which combines the object-orientation presented in Chapter 4 and this theory for sharing and records. With object sharing, the view of the target of a method call as a value-result parameter, whose value is updated to reflect changes carried out by the method, becomes unnecessary, since changes are reflected directly in the objects, not in a copy. Moreover, the kinds of assignments (of values and pointers) and changes in interpretations of λ-abstractions for methods resolutions allows us to have different semantics with copy or reference for object-orientation.

This work also introduces laws for object-oriented programs proved to be sound in the UTP semantic framework. As in [Kas05], here, we also pursue modularity and decoupling, but we concentrate on object-oriented constructs, and moreover we introduce laws and their proofs of soundness. We have presented some laws related to basic constructs such as type tests and laws for method declarations, and due to our representation of methods in the theory, we have some proofs in an algebraic style. The main contribution and novelty related to laws are the simple proofs that are made possible by our modular semantics.

In resume, we:

- **defined** a semantics for object-orientation paradigm in the UTP;
- **integrated** an independent theory for sharing and records to object-orientation where object-oriented programs with copy or reference semantics can be characterized;
- **proved** general laws for object-orientation which are valid for copy or reference semantics.

One of the major results expected by the project of which this thesis is part is to produce a hybrid theory involving classes and processes to describe the semantics in the UTP for languages like *OhCircus* [CSW05] or TCOZ [MD98]. That is, a plan to combine our theory (with copy or reference semantics) with that of CSP processes already introduced in the reference book. The result of this integration, for example, can give rise to a new generation of laws relating classes and processes which can be proved at semantics level and be valid for many different languages as the laws we have presented in this thesis. Our work is a step towards the definition of a semantics for *OhCircus*, an object-oriented combination of Z and CSP.

7.2 Next Steps

The next steps in the thesis development are:

- a better characterisation of open and closed systems to allow
- the description of refinement, or refactoring, presented as
- laws for object-oriented programs.

The first development in this direction is represented by Chapter 6 where we started to characterise open and closed system and provided some laws. Next, we might present the general notion of refinement in the UTP in the context of our theory in order to verify that, as in the other theories, refinement is defined by the logical implication.

Besides these steps, and since we have now an integrated theory capable of handling pointers, we might be able to verify that some of the laws considered for copy semantics are not sound in a context where pointers are handled.

7.3 Schedule

In Table 7.1 we highlight our activities to achieve the thesis' conclusion.

	Month/Year		
Task	[May-Jul]08	[Ago-Oct]08	[Nov]08-[Feb]09
A.1	√		
A.2		√	
A.3			√

Table 7.1: Timetable.

A.1: Revision of the paper submitted to FACJ to resubmission to Theoretical Computer Science. Introduction of a refinement notion and extension of the set of laws already considered;

A.2: Extend the set of examples related to reference semantics, and finalize the selection and proofs for laws for object-oriented programs considering refinement;

A.3: Thesis finalization stage, write missing parts and omissions detected after thesis's proposal and new developments.

In parallel, we expect to be revising and submitting parts of the thesis as papers to conferences, symposiums and journals, as usual. As well as participating of events related to our research area.

Our main goal is to finalize in time with the expected quality. Surely it will be a challenging and laborious task which is up to us. To perform these remaining tasks we will request 6 months of additional time.

7.4 Future Works

There are still many issues to explore in the formalisation using the UTP, and to extend in this object-oriented theory. In fact the formalization of object-oriented languages pose many problems as considered in [LLM06a]. In the sequel we list some of our guidelines of future work.

7.4.1 Features Set Extension

Features like visibility mechanisms, exception handling, concurrency and garbage collection are important candidates in the future extensions of our theory. With an more expressive set of features in the object-oriented theory, we could use this theory to model more realistic programming languages, or object-oriented specification languages like Object-Z [Smi00]. With the introduction of sharing, an important feature of object-oriented programs we have already given a big step in the direction of representing a language like Java in the UTP, but some other features would be required for different applications. For example, in hard-real-time applications the garbage collection must be carefully handled.

7.4.2 Refactorings

The set of laws could also be extended and composed to handle more general laws of programming also known as refactorings. The focus of this thesis is not refactoring itself, the laws are used to allow us to verify theories soundness and their simplicity to carry out proofs. An extensive set of laws and their composition would be desired to allow descriptions of more elaborated refactorings as those proposed by Fowler [FBB+99] and proved by Cornélio [Cor04]. For example, since our

theory provides a reference semantics, some refactorings proposed by Fowler and not considered by Cornélio should also be handled.

7.4.3 Mechanization

In fact the mechanization of the formalism could be auxiliary in laws proofs, and speed up not only the use of such formalism but also to allow an mechanization of the integration between theories. The construction of a tool, or the formalisation of theories of UTP in theorem provers like ProofPower of PVS will contribute to improve the use of the UTP to formalise different paradigms and thus popularize its use as a semantics background.

Appendix A

Theory of Object-Orientation

A.1 Observational Variables

Class Names
$cls : \mathbb{P}\ name.$

Subtype Relation
$sc : name \mapsto name.$

Attribute Information
$atts : name \mapsto (name \mapsto Type).$
where

$$Type := \{\mathbb{B}, \mathbb{Z}\} \cup cls$$

Methods
Methods texts are recorded in high-order-variables in the form

$$\textbf{var}\ m : proc = \texttt{pds}$$

where `pds` is a parametrised command, as in the example below.

$$\texttt{valres self:Object} \bullet \texttt{mc} \lhd \texttt{self is C} \rhd (\texttt{mb} \lhd \texttt{self is B} \rhd (\texttt{ma} \lhd \texttt{self is A} \rhd \bot_{oo}))$$

A.2 Healthiness Conditions

OO1 $P = P \wedge \textbf{\textit{Object}} \in cls$

OO2 $P = P \wedge \mathrm{dom}\, sc = cls \setminus \{\textbf{\textit{Object}}\}$

OO3 $P = P \land \forall\, C : \mathrm{dom}\, sc \bullet (C, \textbf{\textit{Object}}) \in sc^+$

OO4 $P = P \land \mathrm{dom}\, atts = cls$

OO5 $P = P \land \forall\, C_1, C_2 : \mathrm{dom}\, atts \bullet C_1 \neq C_2 \land \mathrm{dom}(atts(C_1)) \cap \mathrm{dom}(atts(C_2)) = \emptyset$

OO6 $P = P \land \mathrm{ran}(\bigcup \mathrm{ran}\, atts) \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup cls$

OO7 $P = P \land \textbf{\textit{Object}} \in cls'$

OO8 $P = P \land \mathrm{dom}\, sc' = cls' \setminus \{\textbf{\textit{Object}}\}$

OO9 $P = P \land \forall\, C : \mathrm{dom}\, sc' \bullet (C, \textbf{\textit{Object}}) \in sc'^+$

OO10 $P = P \land \mathrm{dom}\, atts' = cls'$

OO11 $P = P \land \forall\, C_1, C_2 : \mathrm{dom}\, atts' \bullet C_1 \neq C_2 \land \mathrm{dom}(atts'(C_1)) \cap \mathrm{dom}(atts'(C_2)) = \emptyset$

OO12 $P = P \land \mathrm{ran}(\bigcup \mathrm{ran}\, atts') \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup cls'$

OO13 $P = P;\ \mathit{I\!I}_{oo}$

Other Definitions

$\mathbf{OOI} \equiv \mathbf{OO1} \circ \mathbf{OO2} \circ \mathbf{OO3} \circ \mathbf{OO4} \circ \mathbf{OO5} \circ \mathbf{OO6}$

$\mathit{I\!I}_{oo} \mathrel{\widehat{=}} \mathbf{OOI}(\mathit{I\!I})$

$\mathbf{OO} \equiv \mathbf{OO1} \circ \mathbf{OO2} \circ \mathbf{OO3} \circ \mathbf{OO4} \circ \mathbf{OO5} \circ \mathbf{OO6} \circ \mathbf{OO7} \circ \mathbf{OO8} \circ \mathbf{OO9} \circ \mathbf{OO10} \circ \mathbf{OO11} \circ \mathbf{OO12}$

$\bot_{oo} \mathrel{\widehat{=}} \mathbf{OO}(\bot)$

A.3 Declarations

Class Introduction

$$\textbf{class}\ A\ \textbf{extends}\ B \mathrel{\widehat{=}} \mathbf{OO}\left(\begin{pmatrix} A \notin \mathit{Type} \land \\ B \in cls \end{pmatrix} \vdash \begin{pmatrix} cls' = cls \cup \{A\} \land \\ sc' = sc \cup \{A \mapsto B\} \land \\ atts' = atts \cup \{A \mapsto \emptyset\} \land \\ w' = w \end{pmatrix}\right)$$

$\textbf{where}\ w = in\alpha(\textbf{class}\ A\ \textbf{extends}\ B) \setminus \{\, cls,\ sc,\ atts\}$

or

$\textbf{class}\ A = \textbf{class}\ A\ \textbf{extends}\ \textbf{\textit{Object}}$

Attribute Introduction

$$\textbf{att}\ A\ x : T \mathrel{\widehat=} \textbf{OO}\left(\begin{pmatrix} A \in cls \land \\ x \notin \operatorname{dom} \mathcal{C}(atts, cls) \land \\ T \in Type \end{pmatrix} \vdash \begin{pmatrix} atts' = atts \oplus \{A \mapsto (atts(A) \cup \{x \mapsto T\})\} \land \\ w' = w \end{pmatrix}\right)$$

where $w = in\alpha(\textbf{att}\ A\ x : T) \setminus \{atts\}$
and $\mathcal{C}(amap, cset) = \bigcup\{N : cset \bullet amap\ N\}$,
$amap$ is an attribute mapping, and $cset$ is class set.

Closure of attributes

$$\mathcal{U}(amap, smap, N) = \bigcup amap(\!|\ smap^*(\!|\ \{N\}\ |\!)\ |\!)$$

Method Introduction

$$\textbf{meth}\ A\ m = (pds \bullet p) \mathrel{\widehat=} \textbf{OO}\left(\textbf{var}\ m;\ \begin{pmatrix} A \in cls \land \\ \forall\, t \in types(pds) \bullet t \in Type \end{pmatrix} \vdash \begin{pmatrix} m' = \mathtt{pds}_e \bullet (\mathtt{p} \lhd \mathtt{self\ is\ A} \rhd \bot_{oo}) \land \\ w' = w \end{pmatrix}\right)$$

provided $m \notin \alpha(\textbf{meth}\ A\ m = (pds \bullet p))$
where $\mathtt{pds}_e = \mathtt{valres\ self{:}Object;\ pds}$ **and** $w = in\alpha(\textbf{meth}\ A\ m = (pds \bullet p))$

Method Redefinition

$$\textbf{meth}\ A\ m = (pds \bullet p) \mathrel{\widehat=} \textbf{OO}\left(\begin{pmatrix} A \in cls \land \\ \exists\, \mathtt{q} \bullet m = \mathtt{pds}_e \bullet \mathtt{q} \end{pmatrix} \vdash \left(\exists\, \mathtt{q} \bullet \begin{pmatrix} m = \mathtt{pds}_e \bullet \mathtt{q} \land \\ m' = \mathtt{pds}_e \bullet join(A, \mathtt{p}, \mathtt{q}) \land \\ w' = w \end{pmatrix}\right)\right)$$

provided $m \in \alpha(\textbf{meth}\ A\ m = (pds \bullet p))$
where $\mathtt{pds}_e = \mathtt{valres\ self{:}Object;\ pds}$, $w = in\alpha(\textbf{meth}\ A\ m = (pds \bullet p)) \setminus \{m\}$
and

$join(A, \mathtt{p}, \bot_{oo}) = \mathtt{p} \lhd \mathtt{self\ is\ A} \rhd \bot_{oo}$

$$join(A, \mathtt{p}, \mathtt{q}_l \lhd \mathtt{self\ is\ B} \rhd \mathtt{q}_r) = \begin{cases} \mathtt{p} \lhd \mathtt{self\ is\ A} \rhd (\mathtt{q}_l \lhd \mathtt{self\ is\ B} \rhd \mathtt{q}_r), & \text{if } A \prec B \\ \mathtt{q}_l \lhd \mathtt{self\ is\ B} \rhd join(A, \mathtt{p}, \mathtt{q}_r) & \text{, otherwise} \end{cases}$$

$join(A, \mathtt{p}, \mathtt{q}) = \bot_{oo}$, *for programs* $\mathtt{q}$ *of every other form*

A.4 Abstractions

Call by Value

$$(\textbf{val}\ v : T \bullet p) \mathrel{\widehat=} (\lambda\, w : T \bullet (\textbf{var}\ v : T;\ v := w;\ p;\ \textbf{end}\ v : T))$$

Call by Result

$$(\textbf{res}\ v : T \bullet p) \mathrel{\widehat=} (\lambda\, w : \mathcal{N} \bullet (\textbf{var}\ v : T;\ p;\ w := v;\ \textbf{end}\ v : T))$$

Call by Value-Result

$$(\mathbf{valres}\ v : T \bullet p) \mathrel{\widehat{=}} (\lambda\, w : \mathcal{N} \bullet (\mathbf{var}\ v : T;\ v := w;\ p;\ w := v;\ \mathbf{end}\ v : T))$$

$$(\lambda\, x : \mathcal{N} \bullet p)(y) \mathrel{\widehat{=}} p\,[y, y', yt, yt'/x, x', xt, xt']$$

A.5 Variables

Declaration

$$\mathbf{var}\ x : T \mathrel{\widehat{=}} \mathbf{OO}(\{T \in \mathit{Type}\}_\perp;\ \mathbf{var}\ xt, x;\ \mathbf{true} \vdash xt' = T \land x' \in \mathcal{V}(T) \land w' = w)$$

$$\mathbf{provided}\ x \notin \mathit{in}\alpha(\mathbf{var}\ x : T)\ \mathbf{and}\ w = \mathit{in}\alpha(\mathbf{var}\ x : T)$$

Undeclaration

$$\mathbf{end}\ x : T \mathrel{\widehat{=}} \mathbf{OO}(\mathbf{end}\ x, xt)$$

A.6 Expressions

BNF

$$\begin{array}{rcl} e & ::= & v \mid le \mid \mathbf{new}\ N \mid e\ \mathbf{is}\ N \mid (N)e \mid f(e) \mid \mathbf{null} \\ le & ::= & x \mid \mathbf{self} \mid le.x \end{array}$$

Table A.1: BNF for object-oriented expressions.

Well-definedness

Primitive Values

$$\begin{array}{rcl} \mathcal{D}((\mathbb{B}, v)) & \mathrel{\widehat{=}} & v \in \mathbb{B} \\ \mathcal{D}((\mathbb{Z}, v)) & \mathrel{\widehat{=}} & v \in \mathbb{Z} \end{array}$$

Objects

$$\begin{array}{rcl} \mathcal{D}((T, \mathbf{null})) & \mathrel{\widehat{=}} & T \in \mathit{cls} \\ \mathcal{D}((T, v)) & \mathrel{\widehat{=}} & T \in \mathit{cls} \land (T, v) \in \mathcal{V}(T) \end{array}$$

Variables

$$\begin{array}{rcl} \mathcal{D}(x) & \mathrel{\widehat{=}} & xt \in \mathit{Type} \\ \mathcal{D}(\mathbf{self}) & \mathrel{\widehat{=}} & \mathbf{self}t \in \mathit{cls} \end{array}$$

Attribute Accesses

$$\mathcal{D}(le.x) \mathrel{\widehat{=}} \mathcal{D}(le) \land le_t \in \mathit{cls} \land le_v \neq \mathbf{null} \land x \in \operatorname{dom} le_v$$

Typing

$$\begin{array}{rcl} \mathcal{D}(\textbf{new } N) & \widehat{=} & N \in cls \\ \mathcal{D}(e \textbf{ is } N) & \widehat{=} & \mathcal{D}(e) \land N \in cls \\ \mathcal{D}((N)e) & \widehat{=} & \mathcal{D}(e) \land N \in cls \land e_t \preceq N \end{array}$$

Remainder

$$\mathcal{D}(x \% y) \quad \widehat{=} \quad \mathcal{D}(x) \land \mathcal{D}(y) \land x_t = \mathbb{Z} \land y_t = \mathbb{Z} \land y_v \neq 0$$

Object Creation

$$\textbf{new } N \mathrel{\widehat{=}} \left(N, \left\{ \begin{pmatrix} x : \operatorname{dom} map; \\ t : Type; \\ v : \{\, T : Type;\ i : T \bullet i \,\} \end{pmatrix} \mid \left(\begin{array}{l} \begin{pmatrix} map(x) = \mathbb{B} \land \\ t = \mathbb{B} \land \\ v = \textbf{false} \end{pmatrix} \\ \lor \\ \begin{pmatrix} map(x) = \mathbb{Z} \land \\ t = \mathbb{Z} \land \\ v = 0 \end{pmatrix} \\ \lor \\ \left(\exists\, T : cls \bullet \begin{pmatrix} map(x) = T \land \\ t = T \land \\ v = \textbf{null} \end{pmatrix} \right) \end{array} \right) \bullet x \mapsto (t, v) \right\} \right)$$

where $map = \mathcal{U}(atts, sc, N)$

Type Test

$$e \textbf{ is } N \mathrel{\widehat{=}} (\mathbb{B}, e_t \preceq N)$$

Type Cast

$$(N)e \ \mathrel{\widehat{=}} e$$

Attribute Access

$$le.x \mathrel{\widehat{=}} le_v(x)$$

A.7 Commands

BNF

$$c ::= le := e \mid I\!I \mid \textbf{var}\ x : T \mid \textbf{end}\ x : T \mid c_1 \lhd e \rhd c_2 \mid c_1;\ c_2 \mid \mu\, X \bullet F(X) \mid le.m(e)$$

Table A.2: BNF for object-oriented commands.

Well-definedness

Assignment to variables
$\mathcal{D}(x := e) \ \widehat{=}\ \mathcal{D}(x) \wedge \mathcal{D}(e) \wedge e_t \preceq xt$

Assignment to attributes
$\mathcal{D}(le.x := e) \ \widehat{=}\ \mathcal{D}(le.x) \wedge \mathcal{D}(e) \wedge e_t \preceq \mathcal{U}(atts, sc, le_t)(x)$

Conditional
$\mathcal{D}(P \lhd e \rhd Q) \ \widehat{=}\ \mathcal{D}(e) \wedge e_t = \mathbb{B}$

Method call
$\mathcal{D}(le.m(e)) \ \widehat{=}\ \mathcal{D}(le) \wedge le_v \neq \textbf{null} \wedge compatible(le, m) \wedge e_t \preceq T$

provided $\exists\, \mathtt{p} \bullet m = (\mathtt{val\ x{:}T} \bullet \mathtt{p})$,
where $compatible(le, m) \mathrel{\widehat{=}} \exists\, \mathtt{pds}, \mathtt{p} \bullet m = (\mathtt{pds} \bullet \mathtt{p}) \wedge le_t \in scan(\mathtt{p})$
with
$scan(\bot_{oo}) = \{\}$
$scan(\mathtt{p}_l \lhd \mathtt{self\ is\ A} \rhd \mathtt{p}_r) = \{B : cls \mid B \preceq A\} \cup scan(\mathtt{p}_r)$

$\mathcal{D}(le.m(y)) \ \widehat{=}\ \mathcal{D}(le) \wedge le_v \neq \textbf{null} \wedge compatible(le, m) \wedge sdisjoint(le, y) \wedge T \preceq y_t$
provided $\exists\, \mathtt{p} \bullet m = (\mathtt{res\ x{:}T} \bullet \mathtt{p})$

$\mathcal{D}(le.m(z)) \ \widehat{=}\ \mathcal{D}(le) \wedge le_v \neq \textbf{null} \wedge compatible(le, m) \wedge sdisjoint(le, z) \wedge T = z_t$
provided $\exists\, \mathtt{p} \bullet m = (\mathtt{valres\ x{:}T} \bullet \mathtt{p})$

Assignment of Variables

$x := e \mathrel{\widehat{=}} \mathbf{OO}(\mathcal{D}(x := e) \vdash x' = e \wedge w' = w)$

where $w = in\alpha(x := e) \setminus \{x\}$

Assignment of Attributes

$le.x := e \mathrel{\widehat{=}} \mathbf{OO}(\mathcal{D}(le.x := e) \vdash le' = (le_t, le_v \oplus \{x \mapsto e\}) \wedge w' = w)$

where $w = in\alpha(le.x := e) \setminus \alpha(le)$

Conditional

$$P \lhd e \rhd Q \mathrel{\widehat{=}} \mathbf{OO}(\mathcal{D}(P \lhd e \rhd Q) \wedge ((e_v \wedge P) \vee (\neg e_v \wedge Q)))$$

Simple Recursion

$$\textbf{meth}\ A\ m = \mu\, X \bullet \big(\ pds \bullet F(X)\ \big)$$

Mutual Recursion

$$\textbf{meth}\ A\ m, B\ n = \mu\, X, Y \bullet \big(\ pds_m \bullet F(X, Y), pds_n \bullet G(X, Y)\ \big)$$

Method Call

$$le.m(args) \mathrel{\widehat{=}} \mathbf{OO}(\{\mathcal{D}(le.m(args))\}_\perp;\ (pds_e \bullet p)(le, args))$$

where $m = \mathtt{pds}_e \bullet \mathtt{p}$

Appendix B

Healthiness Condition Laws

B.1 Closedness of OO HCs

Law 6. <**OO1**-*idempotent*>

$$\mathbf{OO1} \circ \mathbf{OO1} = \mathbf{OO1}$$

Proof.

$\mathbf{OO1} \circ \mathbf{OO1}(P)$ *[composition, **OO1**]*
$= \mathbf{OO1}(\textbf{\textit{Object}} \in cls \wedge P)$ *[**OO1**, propositional calculus]*
$= \textbf{\textit{Object}} \in cls \wedge P$ *[**OO1**]*
$= \mathbf{OO1}(P)$

□

Law 7. <**OO1**-∧-*closure*>

$\mathbf{OO1}(P \wedge Q) = P \wedge Q$, *provided* P *and* Q *are* **OO1** *healthy.*

Proof.

$\mathbf{OO1}(P \wedge Q)$ *[**OO1**]*
$= \textbf{\textit{Object}} \in cls \wedge P \wedge Q$ *[propositional calculus]*
$= \textbf{\textit{Object}} \in cls \wedge P \wedge \textbf{\textit{Object}} \in cls \wedge Q$ *[**OO1**]*
$= \mathbf{OO1}(P) \wedge \mathbf{OO1}(Q)$ *[assumption]*
$= P \wedge Q$

□

Law 8. <**OO1**-∨-*closure*>

$\mathbf{OO1}(P \vee Q) = P \vee Q$, *provided* P *and* Q *are* **OO1** *healthy.*

Proof.
Similar to that of **OO1**-∧-closure.

□

Law 9. <**OO1**-$\_ \lhd \_ \rhd \_$-*closure*>

$\mathbf{OO1}(P \lhd b \rhd Q) = P \lhd b \rhd Q$, *provided* P *and* Q *are* **OO1** *healthy.*

Proof.

$$
\begin{array}{ll}
& \mathbf{OO1}(P \lhd b \rhd Q) \\
= & \mathbf{Object} \in cls \wedge (P \lhd b \rhd Q) \\
= & (\mathbf{Object} \in cls \wedge P) \lhd b \rhd (\mathbf{Object} \in cls \wedge Q) \\
= & \mathbf{OO1}(P) \lhd b \rhd \mathbf{OO1}(Q) \\
= & P \lhd b \rhd Q
\end{array}
$$

[OO1] *[conditional-disjunction]* *[OO1]* *[assumption]*

□

Law 10. <**OO1**-; -*closure*>

$\mathbf{OO1}(P;\ Q) = P;\ Q$, *provided* P *and* Q *are* **OO1** *healthy.*

Proof.

$$
\begin{array}{ll}
& \mathbf{OO1}(P;\ Q) \\
= & \mathbf{OO1}(\mathbf{OO1}(P);\ \mathbf{OO1}(Q)) \\
= & ((\mathbf{Object} \in cls \wedge P);\ (\mathbf{Object} \in cls \wedge Q)) \wedge \mathbf{Object} \in cls \\
= & \exists\, cls_0, v_0 \bullet \begin{pmatrix} \mathbf{Object} \in cls \wedge P[cls_0, v_0/cls', v'] \wedge \\ \mathbf{Object} \in cls_0 \wedge Q[cls_0, v_0/cls, v] \end{pmatrix} \wedge \mathbf{Object} \in cls \\
= & \exists\, cls_0, v_0 \bullet \begin{pmatrix} \mathbf{Object} \in cls \wedge P[cls_0, v_0/cls', v'] \wedge \\ \mathbf{Object} \in cls_0 \wedge Q[cls_0, v_0/cls, v] \end{pmatrix} \\
= & \mathbf{OO1}(P);\ \mathbf{OO1}(Q) \\
= & P;\ Q
\end{array}
$$

[assumption] *[OO1]* *[sequence]* *[propositional calculus]* *[sequence]* *[assumption]*

□

Law 11. <**OO1**-μ-*closure*>

$\mathbf{OO1}(\mu\, X \bullet F(X)) = \mu\, X \bullet F(X)$, *provided* $F(X)$ *is* **OO1** *healthy.*

Proof.

$$
\begin{array}{ll}
& \mathbf{OO1}(\mu\, X \bullet F(X)) \\
= & \mathbf{Object} \in cls \wedge \sqcap\{X \mid X \Rightarrow F(X)\} \\
= & \sqcap\{X \wedge \mathbf{Object} \in cls \mid X \Rightarrow \mathbf{OO1}(F(X))\} \\
= & \sqcap\{X \wedge \mathbf{Object} \in cls \mid X \Rightarrow (\mathbf{Object} \in cls \wedge F(X))\} \\
= & \sqcap\{X \wedge \mathbf{false} \mid X \Rightarrow (\mathbf{false} \wedge F(X))\} \vee \sqcap\{X \wedge \mathbf{true} \mid X \Rightarrow (\mathbf{true} \wedge F(X))\} \\
= & \sqcap\{\mathbf{false} \mid X \Rightarrow \mathbf{false}\} \vee \sqcap\{X \mid X \Rightarrow F(X)\} \\
= & \mathbf{false} \vee \sqcap\{X \mid X \Rightarrow F(X)\} \\
= & \sqcap\{X \mid X \Rightarrow F(X)\} \\
= & \mu\, X \bullet F(X)
\end{array}
$$

*[μ,**OO1**]* *[**L3** of relations, assumption]* *[**OO1**, $F(X)$ is healthy]* *[case analysis]* *[propositional calculus]* *[$\sqcap$-definition]* *[disjunction unit- **false** $\vee\ A = A$]* *[μ]*

□

Law 12. <**OO2**-*idempotent*>

$$\mathbf{OO2} \circ \mathbf{OO2} = \mathbf{OO2}$$

Proof.
Similar to that of **OO1**-idempotent.

□

Law 13. <**OO2**-$\wedge$-*closure*>

$\mathbf{OO2}(P \wedge Q) = P \wedge Q$, *provided* P *and* Q *are* **OO2** *healthy.*

Proof.
Similar to that of **OO1**-$\wedge$-closure.

□

Law 14. <**OO2**-$\vee$-*closure*>

$\mathbf{OO2}(P \vee Q) = P \vee Q$, *provided* P *and* Q *are* **OO2** *healthy.*

Proof.
Similar to that of **OO1**-$\wedge$-closure.

□

Law 15. <**OO2**-$\_ \lhd \_ \rhd \_$-*closure*>

$\mathbf{OO2}(P \lhd b \rhd Q) = P \lhd b \rhd Q$, *provided* P *and* Q *are* **OO2** *healthy.*

Proof.
Similar to that of **OO1**-$\_ \lhd \_ \rhd \_$-closure.

□

Law 16. <**OO2**-; -*closure*>

$\mathbf{OO2}(P;\ Q) = P;\ Q$, *provided* P *and* Q *are* **OO2** *healthy.*

Proof.

$$
\begin{aligned}
& \mathbf{OO2}(P;\ Q) && \textit{[assumption]}\\
&= \mathbf{OO2}(\mathbf{OO2}(P);\ \mathbf{OO2}(Q)) && \textit{[OO2]}\\
&= \begin{pmatrix} \left(\begin{pmatrix} \mathrm{dom}\ sc = cls \setminus \{\textbf{\textit{Object}}\} \wedge \\ P \end{pmatrix} ;\ \begin{pmatrix} \mathrm{dom}\ sc = cls \setminus \{\textbf{\textit{Object}}\} \wedge \\ Q \end{pmatrix} \right) \\ \wedge \\ \mathrm{dom}\ sc = cls \setminus \{\textbf{\textit{Object}}\} \end{pmatrix} && \textit{[sequence]}\\
&= \begin{pmatrix} \exists\, cls_0, sc_0, v_0 \bullet \left(\begin{pmatrix} \mathrm{dom}\ sc = cls \setminus \{\textbf{\textit{Object}}\} \wedge \\ P[cls_0, sc_0, v_0 / cls', sc', v'] \end{pmatrix} \wedge \begin{pmatrix} \mathrm{dom}\ sc_0 = cls_0 \setminus \{\textbf{\textit{Object}}\} \wedge \\ Q[cls_0, sc_0, v_0 / cls, sc, v] \end{pmatrix} \right) \\ \wedge \\ \mathrm{dom}\ sc = cls \setminus \{\textbf{\textit{Object}}\} \end{pmatrix} && \textit{[propositional calculus]}\\
&= \exists\, cls_0, sc_0, v_0 \bullet \left(\begin{pmatrix} \mathrm{dom}\ sc = cls \setminus \{\textbf{\textit{Object}}\} \wedge \\ P[cls_0, sc_0, v_0 / cls', sc', v'] \end{pmatrix} \wedge \begin{pmatrix} \mathrm{dom}\ sc_0 = cls_0 \setminus \{\textbf{\textit{Object}}\} \wedge \\ Q[cls_0, sc_0, v_0 / cls, sc, v] \end{pmatrix} \right) && \textit{[sequence]}\\
&= \mathbf{OO2}(P);\ \mathbf{OO2}(Q) && \textit{[assumption]}\\
&= P;\ Q
\end{aligned}
$$

□

Law 17. <**OO2**-μ-*closure*>

$\mathbf{OO2}(\mu\, X \bullet F(X)) = \mu\, X \bullet F(X)$, *provided* $F(X)$ *is* **OO2** *healthy.*

Proof.
Similar to that of **OO1**-μ-closure.

□

Law 18. <**OO3**-*idempotent*>

$$\mathbf{OO3} \circ \mathbf{OO3} = \mathbf{OO3}$$

Proof.
Similar to that of **OO1**-idempotent.

□

Law 19. <**OO3**-$\wedge$-*closure*>

$$\mathbf{OO3}(P \wedge Q) = P \wedge Q, \text{ provided } P \text{ and } Q \text{ are } \mathbf{OO3} \text{ healthy.}$$

Proof.
Similar to that of **OO1**-$\wedge$-closure.

□

Law 20. <**OO3**-$\vee$-*closure*>

$$\mathbf{OO3}(P \vee Q) = P \vee Q, \text{ provided } P \text{ and } Q \text{ are } \mathbf{OO3} \text{ healthy.}$$

Proof.
Similar to that of **OO1**-$\wedge$-closure.

□

Law 21. <**OO3**-$\_ \lhd \_ \rhd \_$-*closure*>

$$\mathbf{OO3}(P \lhd b \rhd Q) = P \lhd b \rhd Q, \text{ provided } P \text{ and } Q \text{ are } \mathbf{OO3} \text{ healthy.}$$

Proof.
Similar to that of **OO1**-$\_ \lhd \_ \rhd \_$-closure.

□

Law 22. <**OO3**-; -*closure*>

$$\mathbf{OO3}(P;\ Q) = P;\ Q, \text{ provided } P \text{ and } Q \text{ are } \mathbf{OO3} \text{ healthy.}$$

Proof.

$$\begin{aligned}
&\mathbf{OO3}(P;\ Q) && \text{[assumption]}\\
&= \mathbf{OO3}(\mathbf{OO3}(P);\ \mathbf{OO3}(Q)) && \text{[OO3]}\\
&= \begin{pmatrix} \left(\begin{pmatrix}\forall C : \mathrm{dom}\, sc \bullet (C, \textbf{\textit{Object}}) \in sc^+\\ \wedge\ P\end{pmatrix};\ \begin{pmatrix}\forall C : \mathrm{dom}\, sc \bullet (C, \textbf{\textit{Object}}) \in sc^+\\ \wedge\ Q\end{pmatrix}\right)\\ \wedge\\ \forall C : \mathrm{dom}\, sc \bullet (C, \textbf{\textit{Object}}) \in sc^+ \end{pmatrix} && \text{[sequence]}\\
&= \begin{pmatrix} \exists sc_0, v_0 \bullet \left(\begin{pmatrix}\forall C : \mathrm{dom}\, sc \bullet (C, \textbf{\textit{Object}}) \in sc^+\\ \wedge\ P[sc_0, v_0/sc', v']\end{pmatrix} \wedge \begin{pmatrix}\forall C : \mathrm{dom}\, sc_0 \bullet (C, \textbf{\textit{Object}}) \in sc_0^+\\ \wedge\ Q[sc_0, v_0/sc, v]\end{pmatrix}\right)\\ \wedge\\ \forall C : \mathrm{dom}\, sc \bullet (C, \textbf{\textit{Object}}) \in sc^+ \end{pmatrix} && \text{[propositional calculus]}\\
&= \exists sc_0, v_0 \bullet \left(\begin{pmatrix}\forall C : \mathrm{dom}\, sc \bullet (C, \textbf{\textit{Object}}) \in sc^+\\ \wedge\ P[sc_0, v_0/sc', v']\end{pmatrix} \wedge \begin{pmatrix}\forall C : \mathrm{dom}\, sc_0 \bullet (C, \textbf{\textit{Object}}) \in sc_0^+\\ \wedge\ Q[sc_0, v_0/sc, v]\end{pmatrix}\right) && \text{[sequence]}\\
&= \mathbf{OO3}(P);\ \mathbf{OO3}(Q) && \text{[assumption]}\\
&= P;\ Q
\end{aligned}$$

□

Law 23. <**OO3**-μ-*closure*>

$$\mathbf{OO3}(\mu\, X \bullet F(X)) = \mu\, X \bullet F(X),\ \textit{provided}\ F(X)\ \textit{is}\ \mathbf{OO3}\ \textit{healthy.}$$

Proof.
Similar to that of **OO1**-μ-closure. □

Law 24. <**OO4**-*idempotent*>

$$\mathbf{OO4} \circ \mathbf{OO4} = \mathbf{OO4}$$

Proof.
Similar to that of **OO1**-idempotent. □

Law 25. <**OO4**-$\wedge$-*closure*>

$$\mathbf{OO4}(P \wedge Q) = P \wedge Q,\ \textit{provided}\ P\ \textit{and}\ Q\ \textit{are}\ \mathbf{OO4}\ \textit{healthy.}$$

Proof.
Similar to that of **OO1**-$\wedge$-closure. □

Law 26. <**OO4**-$\vee$-*closure*>

$$\mathbf{OO4}(P \vee Q) = P \vee Q,\ \textit{provided}\ P\ \textit{and}\ Q\ \textit{are}\ \mathbf{OO4}\ \textit{healthy.}$$

Proof.
Similar to that of **OO1**-$\wedge$-closure. □

Law 27. <**OO4**-$\_ \lhd \_ \rhd \_$-*closure*>

$$\mathbf{OO4}(P \lhd b \rhd Q) = P \lhd b \rhd Q,\ \textit{provided}\ P\ \textit{and}\ Q\ \textit{are}\ \mathbf{OO4}\ \textit{healthy.}$$

Proof.
Similar to that of **OO1**-$\_ \lhd \_ \rhd \_$-closure. □

Law 28. <**OO4**-; -*closure*>

$$\mathbf{OO4}(P;\ Q) = P;\ Q,\ \textit{provided}\ P\ \textit{and}\ Q\ \textit{are}\ \mathbf{OO4}\ \textit{healthy.}$$

Proof.

$$\begin{array}{lll}
\mathbf{OO4}(P;\ Q) & & \textit{[assumption]} \\
= \mathbf{OO4}(\mathbf{OO4}(P);\ \mathbf{OO4}(Q)) & & \textit{[OO4]} \\
= \big(\ (\mathrm{dom}\ atts = cls \wedge P);\ (\mathrm{dom}\ atts = cls \wedge Q)\ \big) \wedge \mathrm{dom}\ atts = cls & & \textit{[sequence]} \\
= \left(\begin{array}{l} \exists\, cls_0, atts_0, v_0 \bullet \left(\begin{array}{l} \mathrm{dom}\ atts = cls \wedge P[cls_0, atts_0, v_0 / cls', atts', v'] \wedge \\ \mathrm{dom}\ atts_0 = cls_0 \wedge Q[cls_0, atts_0, v_0 / cls, atts, v] \end{array} \right) \\ \wedge \\ \mathrm{dom}\ atts = cls \end{array} \right) & & \\
 & & \textit{[propositional calculus]}
\end{array}$$

$$= \exists\, cls_0, atts_0, v_0 \bullet \left(\begin{array}{l} \mathrm{dom}\, atts = cls \wedge P[cls_0, atts_0, v_0 / cls', atts', v'] \wedge \\ \mathrm{dom}\, atts_0 = cls_0 \wedge Q[cls_0, atts_0, v_0 / cls, atts, v] \end{array} \right) \quad \textit{[sequence]}$$

$$= \mathbf{OO4}(P);\ \mathbf{OO4}(Q) \quad \textit{[assumption]}$$

$$= P;\ Q$$

□

Law 29. <**OO4**-μ-*closure*>

$$\mathbf{OO4}(\mu\, X \bullet F(X)) = \mu\, X \bullet F(X),\ \textit{provided}\ F(X)\ \textit{is}\ \mathbf{OO4}\ \textit{healthy.}$$

Proof.
Similar to that of **OO1**-μ-closure.

□

Law 30. <**OO5**-*idempotent*>

$$\mathbf{OO5} \circ \mathbf{OO5} = \mathbf{OO5}$$

Proof.
Similar to that of **OO1**-idempotent.

□

Law 31. <**OO5**-$\wedge$-*closure*>

$$\mathbf{OO5}(P \wedge Q) = P \wedge Q,\ \textit{provided}\ P\ \textit{and}\ Q\ \textit{are}\ \mathbf{OO5}\ \textit{healthy.}$$

Proof.
Similar to that of **OO1**-$\wedge$-closure.

□

Law 32. <**OO5**-$\vee$-*closure*>

$$\mathbf{OO5}(P \vee Q) = P \vee Q,\ \textit{provided}\ P\ \textit{and}\ Q\ \textit{are}\ \mathbf{OO5}\ \textit{healthy.}$$

Proof.
Similar to that of **OO1**-$\wedge$-closure.

□

Law 33. <**OO5**-$\_ \lhd \_ \rhd \_$-*closure*>

$$\mathbf{OO5}(P \lhd b \rhd Q) = P \lhd b \rhd Q,\ \textit{provided}\ P\ \textit{and}\ Q\ \textit{are}\ \mathbf{OO5}\ \textit{healthy.}$$

Proof.
Similar to that of **OO1**-$\_ \lhd \_ \rhd \_$-closure.

□

Law 34. <**OO5**-; -*closure*>

$$\mathbf{OO5}(P;\ Q) = P;\ Q,\ \textit{provided}\ P\ \textit{and}\ Q\ \textit{are}\ \mathbf{OO5}\ \textit{healthy.}$$

Proof.

$$
\begin{array}{ll}
& \mathbf{OO5}(P;\ Q) \qquad \textit{[assumption]} \\
= & \mathbf{OO5}(\mathbf{OO5}(P);\ \mathbf{OO5}(Q)) \qquad \textit{[OO5]} \\
= & \left(\begin{array}{l}\left(\begin{array}{l}(\forall\, C_1, C_2 : \mathrm{dom}\ atts \bullet C_1 \neq C_2 \wedge \mathrm{dom}(atts(C_1)) \cap \mathrm{dom}(atts(C_2)) = \emptyset \wedge P);\\ (\forall\, C_1, C_2 : \mathrm{dom}\ atts \bullet C_1 \neq C_2 \wedge \mathrm{dom}(atts(C_1)) \cap \mathrm{dom}(atts(C_2)) = \emptyset \wedge Q)\end{array}\right) \\ \wedge \\ \forall\, C_1, C_2 : \mathrm{dom}\ atts \bullet C_1 \neq C_2 \wedge \mathrm{dom}(atts(C_1)) \cap \mathrm{dom}(atts(C_2)) = \emptyset\end{array}\right) \\
& \qquad \textit{[sequence]} \\
= & \left(\begin{array}{l}\exists\, sc_0, v_0 \bullet \left(\begin{array}{l}\forall\, C_1, C_2 : \mathrm{dom}\ atts \bullet C_1 \neq C_2 \wedge \mathrm{dom}(atts(C_1)) \cap \mathrm{dom}(atts(C_2)) = \emptyset \wedge P[sc_0, v_0/sc', v'] \\ \wedge \\ \forall\, C_1, C_2 : \mathrm{dom}\ atts_0 \bullet C_1 \neq C_2 \wedge \mathrm{dom}(atts_0(C_1)) \cap \mathrm{dom}(atts_0(C_1)) = \emptyset \wedge Q[sc_0, v_0/sc, v]\end{array}\right) \\ \wedge \\ \forall\, C_1, C_2 : \mathrm{dom}\ atts \bullet C_1 \neq C_2 \wedge \mathrm{dom}(atts(C_1)) \cap \mathrm{dom}(atts(C_2)) = \emptyset\end{array}\right) \\
& \qquad \textit{[propositional calculus]} \\
= & \exists\, sc_0, v_0 \bullet \left(\begin{array}{l}\forall\, C_1, C_2 : \mathrm{dom}\ atts \bullet C_1 \neq C_2 \wedge \mathrm{dom}(atts(C_1)) \cap \mathrm{dom}(atts(C_2)) = \emptyset \wedge P[sc_0, v_0/sc', v'] \\ \wedge \\ \forall\, C_1, C_2 : \mathrm{dom}\ atts_0 \bullet C_1 \neq C_2 \wedge \mathrm{dom}(atts_0(C_1)) \cap \mathrm{dom}(atts_0(C_1)) = \emptyset \wedge Q[sc_0, v_0/sc, v]\end{array}\right) \\
& \qquad \textit{[sequence]} \\
= & \mathbf{OO5}(P);\ \mathbf{OO5}(Q) \qquad \textit{[assumption]} \\
= & P;\ Q
\end{array}
$$

□

Law 35. <**OO5**-μ-*closure*>

$\mathbf{OO5}(\mu\, X \bullet F(X)) = \mu\, X \bullet F(X)$, *provided* $F(X)$ *is* **OO5** *healthy.*

Proof.
Similar to that of **OO1**-μ-closure.

□

Law 36. <**OO6**-*idempotent*>

$\mathbf{OO6} \circ \mathbf{OO6} = \mathbf{OO6}$

Proof.
Similar to that of **OO1**-idempotent.

□

Law 37. <**OO6**-$\wedge$-*closure*>

$\mathbf{OO6}(P \wedge Q) = P \wedge Q$, *provided* P *and* Q *are* **OO6** *healthy.*

Proof.
Similar to that of **OO1**-$\wedge$-closure.

□

Law 38. <**OO6**-$\vee$-*closure*>

$\mathbf{OO6}(P \vee Q) = P \vee Q$, *provided* P *and* Q *are* **OO6** *healthy.*

Proof.
Similar to that of **OO1**-$\wedge$-closure.

□

Law 39. <**OO6**-$\_ \lhd \_ \rhd \_$-*closure*>

$$\mathbf{OO6}(P \lhd b \rhd Q) = P \lhd b \rhd Q, \textit{ provided } P \textit{ and } Q \textit{ are } \mathbf{OO6} \textit{ healthy.}$$

Proof.
Similar to that of **OO1**-$\_ \lhd \_ \rhd \_$-closure.

□

Law 40. <**OO6**-; -*closure*>

$$\mathbf{OO6}(P;\ Q) = P;\ Q, \textit{ provided } P \textit{ and } Q \textit{ are } \mathbf{OO6} \textit{ healthy.}$$

Proof.

$$\begin{array}{ll}
& \mathbf{OO6}(P;\ Q) \qquad \textit{[assumption]} \\
= & \mathbf{OO6}(\mathbf{OO6}(P);\ \mathbf{OO6}(Q)) \qquad \textit{[OO6]} \\
= & \begin{pmatrix} ((\mathrm{ran}(\bigcup \mathrm{ran}\ atts) \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup cls \wedge P);\ (\mathrm{ran}(\bigcup \mathrm{ran}\ atts) \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup cls \wedge Q)) \\ \wedge \\ \mathrm{ran}(\bigcup \mathrm{ran}\ atts) \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup cls \end{pmatrix} \\
& \qquad \textit{[sequence]} \\
= & \begin{pmatrix} \exists\, cls_0, atts_0, v_0 \bullet \begin{pmatrix} \mathrm{ran}(\bigcup \mathrm{ran}\ atts) \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup cls \wedge P[cls_0, atts_0, v_0 / cls', atts', v'] \\ \wedge \\ \mathrm{ran}(\bigcup \mathrm{ran}\ atts_0) \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup cls_0 \wedge Q[cls_0, atts_0, v_0 / cls, atts, v] \end{pmatrix} \\ \wedge \\ \mathrm{ran}(\bigcup \mathrm{ran}\ atts) \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup cls \end{pmatrix} \\
& \qquad \textit{[propositional calculus]} \\
= & \exists\, cls_0, atts_0, v_0 \bullet \begin{pmatrix} \mathrm{ran}(\bigcup \mathrm{ran}\ atts) \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup cls \wedge P[cls_0, atts_0, v_0 / cls', atts', v'] \\ \wedge \\ \mathrm{ran}(\bigcup \mathrm{ran}\ atts_0) \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup cls_0 \wedge Q[cls_0, atts_0, v_0 / cls, atts, v] \end{pmatrix} \\
& \qquad \textit{[sequence]} \\
= & \mathbf{OO6}(P);\ \mathbf{OO6}(Q) \qquad \textit{[assumption]} \\
= & P;\ Q
\end{array}$$

□

Law 41. <**OO6**-μ-*closure*>

$$\mathbf{OO6}(\mu\, X \bullet F(X)) = \mu\, X \bullet F(X), \textit{ provided } F(X) \textit{ is } \mathbf{OO6} \textit{ healthy.}$$

Proof.
Similar to that of **OO1**-μ-closure.

□

Proofs for healthiness conditions **OO7**-**OO12** are all similar to the closedness proofs of their correspondent initial variables counterparts healthiness conditions.

B.2 Commutativity of OO HCs

Law 42. <**OO1**-**OO2**-*commutativity*>

$$\mathbf{OO1} \circ \mathbf{OO2} = \mathbf{OO2} \circ \mathbf{OO1}$$

Proof.

$$
\begin{aligned}
& \mathbf{OO1} \circ \mathbf{OO2}(P) && \textit{[composition, } \mathbf{OO2}\textit{]} \\
&= \mathbf{OO1}(\mathrm{dom}\ sc = cls \setminus \{\textbf{\textit{Object}}\} \wedge P) && \textit{[}\mathbf{OO1}\textit{]} \\
&= \textbf{\textit{Object}} \in cls \wedge \mathrm{dom}\ sc = cls \setminus \{\textbf{\textit{Object}}\} \wedge P \\
& && \textit{[propositional calculus]} \\
&= \mathrm{dom}\ sc = cls \setminus \{\textbf{\textit{Object}}\} \wedge \textbf{\textit{Object}} \in cls \wedge P \\
& && \textit{[}\mathbf{OO1}\textit{]} \\
&= \mathrm{dom}\ sc = cls \setminus \{\textbf{\textit{Object}}\} \wedge \mathbf{OO1}(P) && \textit{[}\mathbf{OO2}\textit{]} \\
&= \mathbf{OO2} \circ \mathbf{OO1}(P)
\end{aligned}
$$

□

Law 43. <**OO1**-**OO3**-*commutativity*>

$$\mathbf{OO1} \circ \mathbf{OO3} = \mathbf{OO3} \circ \mathbf{OO1}$$

Proof.
Similar to that of $\mathbf{OO1} \circ \mathbf{OO2}$.

□

Law 44. <**OO1**-**OO4**-*commutativity*>

$$\mathbf{OO1} \circ \mathbf{OO4} = \mathbf{OO4} \circ \mathbf{OO1}$$

Proof.
Similar to that of $\mathbf{OO1} \circ \mathbf{OO2}$.

□

Law 45. <**OO1**-**OO5**-*commutativity*>

$$\mathbf{OO1} \circ \mathbf{OO5} = \mathbf{OO5} \circ \mathbf{OO1}$$

Proof.
Similar to that of $\mathbf{OO1} \circ \mathbf{OO2}$.

□

Law 46. <**OO1**-**OO6**-*commutativity*>

$$\mathbf{OO1} \circ \mathbf{OO6} = \mathbf{OO6} \circ \mathbf{OO1}$$

Proof.
Similar to that of $\mathbf{OO1} \circ \mathbf{OO2}$.

□

Law 47. <**OO2**-**OO3**-*commutativity*>

$$\mathbf{OO2} \circ \mathbf{OO3} = \mathbf{OO2} \circ \mathbf{OO3}$$

Proof.
Similar to that of $\mathbf{OO1} \circ \mathbf{OO2}$.

□

Law 48. <**OO2**-**OO4**-*commutativity*>

$$\mathbf{OO2} \circ \mathbf{OO4} = \mathbf{OO2} \circ \mathbf{OO4}$$

Proof.
Similar to that of **OO1** ∘ **OO2**.

□

Law 49. <**OO2**-**OO5**-*commutativity*>

OO2 ∘ **OO5** = **OO2** ∘ **OO5**

Proof.
Similar to that of **OO1** ∘ **OO2**.

□

Law 50. <**OO2**-**OO6**-*commutativity*>

OO2 ∘ **OO6** = **OO2** ∘ **OO6**

Proof.
Similar to that of **OO1** ∘ **OO2**.

□

Law 51. <**OO3**-**OO4**-*commutativity*>

OO3 ∘ **OO4** = **OO3** ∘ **OO4**

Proof.
Similar to that of **OO1** ∘ **OO2**.

□

Law 52. <**OO3**-**OO5**-*commutativity*>

OO3 ∘ **OO5** = **OO3** ∘ **OO5**

Proof.
Similar to that of **OO1** ∘ **OO2**.

□

Law 53. <**OO3**-**OO6**-*commutativity*>

OO3 ∘ **OO6** = **OO3** ∘ **OO6**

Proof.
Similar to that of **OO1** ∘ **OO2**.

□

Law 54. <**OO4**-**OO5**-*commutativity*>

OO4 ∘ **OO5** = **OO4** ∘ **OO5**

Proof.
Similar to that of **OO1** ∘ **OO2**.

□

Law 55. <**OO4**-**OO6**-*commutativity*>

OO4 ∘ **OO6** = **OO4** ∘ **OO6**

Proof.
Similar to that of **OO1** ∘ **OO2**.

□

Law 56. <**OO5**-**OO6**-*commutativity*>

$$\mathbf{OO5} \circ \mathbf{OO6} = \mathbf{OO5} \circ \mathbf{OO6}$$

Proof.
Similar to that of **OO1** ∘ **OO2**.

□

B.3 Other HCs Laws

In the following law, we use the notation '−' to stand for any variable name. For example, we use $[-_0/-']$ to represent the replacement of all final variables, say x', with its corresponding version with a 0 subscript, x_0.

Law 57. <**OO7-12**,**OO13**-*equivalence*>

$$\mathbf{OO13} = \mathbf{OO7} \circ \mathbf{OO8} \circ \mathbf{OO9} \circ \mathbf{OO10} \circ \mathbf{OO11} \circ \mathbf{OO12}$$

Proof.

$$\begin{array}{ll}
& \mathbf{OO13}(P) \qquad \textit{[OO13]} \\
= & P;\ \mathit{II}_{oo} \qquad \textit{[}\mathit{II}_{oo}\textit{]} \\
= & P;\ \mathbf{OOI}(cls' = cls \land sc' = sc \land atts' = atts \land \dots) \qquad \textit{[sequence]} \\
= & \exists -_0 \bullet P[-_0/-'] \land \mathbf{OOI}(cls' = cls \land sc' = sc \land atts' = atts \land \dots)[-_0/-] \\
& \qquad \textit{[OO1-6]} \\
= & \exists -_0 \bullet P[-_0/-'] \land \left(\begin{array}{l} \textbf{\textit{Object}} \in cls \land \\ \mathrm{dom}\ sc = cls \setminus \{\textbf{\textit{Object}}\} \land \\ \forall\, C : \mathrm{dom}\ sc \bullet (C, \textbf{\textit{Object}}) \in sc^+ \land \\ \mathrm{dom}\ atts = cls \land \\ \forall\, C_1, C_2 : \mathrm{dom}\ atts \bullet C_1 \neq C_2 \land \mathrm{dom}(atts(C_1)) \cap \mathrm{dom}(atts(C_2)) = \emptyset \land \\ \mathrm{ran}(\bigcup \mathrm{ran}\ atts) \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup cls \land \\ cls' = cls \land sc' = sc \land atts' = atts \land \dots \end{array}\right)[-_0/-] \\
& \qquad \textit{[substitution]} \\
= & \exists -_0 \bullet P[-_0/-'] \land \left(\begin{array}{l} \textbf{\textit{Object}} \in cls_0 \\ \mathrm{dom}\ sc_0 = cls_0 \setminus \{\textbf{\textit{Object}}\} \land \\ \forall\, C : \mathrm{dom}\ sc_0 \bullet (C, \textbf{\textit{Object}}) \in sc_0^+ \land \\ \mathrm{dom}\ atts_0 = cls_0 \land \\ \forall\, C_1, C_2 : \mathrm{dom}\ atts_0 \bullet \mathrm{dom}(atts_0(C_1)) \cap \mathrm{dom}\ atts_0(C_1)) = \emptyset \land \\ \mathrm{ran}(\bigcup \mathrm{ran}\ atts_0) \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup cls_0 \land \\ cls' = cls_0 \land sc' = sc_0 \land atts' = atts_0 \land \dots \end{array}\right) \\
& \qquad \textit{[}\mathbf{OO7\text{-}12}\textit{, variable values, propositional calculus]} \\
= & \mathbf{OO7} \circ \mathbf{OO8} \circ \mathbf{OO9} \circ \mathbf{OO10} \circ \mathbf{OO11} \circ \mathbf{OO12}(P)
\end{array}$$

□

Appendix C

Theory of Pointers

C.1 Observational Variables

Pointer Machine

$$< A : \mathbb{P} Ad,\ V : Ad \mapsto Value,\ S : Ad \leftrightarrow Ad >$$

where

$$Ad \mathrel{\widehat{=}} (seq\ Label) \backslash \{<>\}$$

Projections

$$A.p \mathrel{\widehat{=}} \{q : Ad \mid p.q \in A\}$$

$$V.p \mathrel{\widehat{=}} \{q : Ad \mid p.q \in \mathrm{dom}\ V \bullet q \mapsto V(p.q)\}$$

Equality

$$p =_v q \equiv A.p = A.q \wedge V.p = V.q$$

$$p =_p q \equiv (p, q) \in S$$

C.2 Healthiness Conditions

HP1 $P = P \wedge \forall\, a_1 : A;\ a_2 : Ad \mid a_2 < a_1 \bullet a_2 \in A$

HP2 $P = P \wedge \mathrm{dom}\ V = term(A)$

HP3 $P = P \wedge \bigwedge\{\ 'x : vars(A) \bullet x =!'x\}$

where $NPV \cap vars(A) = \emptyset$

and $\{A, V, S\} \subseteq NPV$.

$vars(X) \mathrel{\widehat=} \{x : X \bullet x(1)\}$

$$!x \mathrel{\widehat=} \begin{cases} V(x)\,, & \textit{if } x \in term(A) \\ V.x & \textit{, otherwise.} \end{cases} \qquad !x' \mathrel{\widehat=} \begin{cases} V'(x)\,, & \textit{if } x \in term(A') \\ V'.x & \textit{, otherwise.} \end{cases}$$

HP4 $P = P \wedge S \in (A \leftrightarrow A) \wedge S = S^*$

HP5 $P = P \wedge fclos_A\ S$

where $fclos_A\ E \mathrel{\widehat=} \forall\, x, y : Ad \mid (x, y) \in E \bullet \forall\, a : Ad \mid x.a \in A \wedge y.a \in A \bullet (x.a, y.a) \in E$

HP6 $P = P \wedge \forall\, a, b : Ad \bullet (a, b) \in S \wedge a \in \mathrm{dom}\ \ V \Rightarrow b \in \mathrm{dom}\ V \wedge V(a) = V(b)$

HP7-12 correspond to **HP1-6** but restricting the final values for A, V and S.

Other Definitions

HPI $\equiv$ **HP1** $\circ$ **HP2** $\circ$ **HP3** $\circ$ **HP4** $\circ$ **HP5** $\circ$ **HP6**

HP $\equiv$ **HP1** $\circ$ **HP2** $\circ$ **HP3** $\circ$ **HP4** $\circ$ **HP5** $\circ$ **HP6** $\circ$ **HP7** $\circ$ **HP8** $\circ$ **HP9** $\circ$ **HP10** $\circ$ **HP11** $\circ$ **HP12**

C.3 Variables

Declaration

$$\textbf{var}\ x \mathrel{\widehat=} \textbf{HPI} \circ \textbf{HP9} \left(\exists\, v : \mathit{Value} \bullet \begin{pmatrix} A' = A \cup \{x\} \wedge \\ V' = V \oplus \{x \mapsto v\} \wedge \\ S' = S \cup \{x \mapsto x\} \end{pmatrix} \right)$$

provided $x \notin A$

Undeclaration

$$\textbf{end}\ x \mathrel{\widehat{=}} \textbf{HPI} \circ \textbf{HP9} \begin{pmatrix} A' = A \setminus \{x\infty \cup \{x\}\} \land \\ V' = (x\infty \cup \{x\}) \lhd V \land \\ S' = (x\infty \cup \{x\}) \lhd S \rhd (x\infty \cup \{x\}) \end{pmatrix}$$

provided $x \in A$
where $x\infty \mathrel{\widehat{=}} \{a : Ad \bullet x.a\}$

C.4 Commands

Auxiliary Functions

$$share_X(x) \mathrel{\widehat{=}} X(\!|\ \{x\}\ |\!)$$

$$X^{\uparrow} \mathrel{\widehat{=}} \bigcup\{x : X \bullet x\infty\}$$
$$ext_X(x) \mathrel{\widehat{=}} share_X(x)^{\uparrow}$$

Value Assignment

$$x :=_t e \mathrel{\widehat{=}} \textbf{HPI} \circ \textbf{HP9} \begin{pmatrix} A' = A \land \\ V' = V \oplus \{a : share_S(x) \bullet a \mapsto e\} \land \\ S' = S \end{pmatrix}$$

provided $x \in \mathrm{dom}\ V$

$$x :=_i e \mathrel{\widehat{=}} \textbf{HPI} \circ \textbf{HP9} \begin{pmatrix} A' = A \setminus ext_S(x) \land \\ V' = (ext_S(x) \lhd V) \cup \{a : (share_S(x) \setminus ext_S(x)) \bullet a \mapsto e\} \land \\ S' = ext_S(x) \lhd S \rhd ext_S(x) \end{pmatrix}$$

provided $x \in A, x \notin \mathrm{dom}\ V$

Pointer Assignment

$$x :-_t y \mathrel{\widehat{=}} \textbf{HPI} \circ \textbf{HP9} \begin{pmatrix} A' = A \cup \{a : Ad \mid y.a \in A \bullet x.a\} \land \\ V' = (\{x\} \lhd V) \oplus \{a : \mathrm{dom}\ V;\ fa : seq\ Label \mid a = y.fa \bullet x.fa \mapsto V(y.fa)\} \land \\ S' = ((\{x\} \lhd S \rhd \{x\}) \cup \{x \mapsto y\}) \cup \{a : Ad \mid y.a \in A \bullet x.a \mapsto y.a\})^* \end{pmatrix}$$

provided $x \in \mathrm{dom}\ V$

$$x :-_i y \mathrel{\widehat{=}} \textbf{HPI} \circ \textbf{HP9} \begin{pmatrix} A' = (A \setminus x\infty) \cup \{a : Ad \mid y.a \in A \bullet x.a\} \land \\ V' = (x\infty \lhd V) \oplus \{a : \mathrm{dom}\ V;\ fa : seq\ Label \mid a = y.fa \bullet x.fa \mapsto V(y.fa)\} \land \\ S' = (((x\infty \cup \{x\}) \lhd S \rhd (x\infty \cup \{x\})) \cup \{x \mapsto y\} \cup \\ \qquad \{a : Ad \mid y.a \in A \bullet x.a \mapsto y.a\})^* \end{pmatrix}$$

provided $x \in A, x \notin \mathrm{dom}\ V$

Register Creation and Assignment

$$\mathbf{new}(f) \mathrel{\widehat{=}} \{n : name;\ v : Value \mid n \in f \bullet n \mapsto v\}$$

$$x :=_r \mathbf{new}(f) \mathrel{\widehat{=}} \mathbf{HPI} \circ \mathbf{HP9} \begin{pmatrix} A' = (A \setminus x\infty) \cup \{n : name \mid n \in \operatorname{dom} map \bullet x.n\} \wedge \\ V' = ((x\infty \cup \{x\}) \lhd V) \oplus \{n : name \mid n \in \operatorname{dom} map \bullet x.n \mapsto map(n)\} \wedge \\ S' = (x\infty \cup \{x\}) \lhd S \rhd (x\infty \cup \{x\}) \end{pmatrix}$$

provided $x \in A$
where $map = \mathbf{new}(f)$

Appendix D

Integrated Theory

D.1 Observational Variables

Class Names

$cls : \mathbb{P}\ name.$

Subtype Relation

$sc : name \mapsto name.$

Attribute Information

$atts : name \mapsto (name \mapsto Type).$
where

$$Type := \{\mathbb{B}, \mathbb{Z}\} \cup cls$$

Methods

Methods texts are recorded in high-order-variables in the form

$$\textbf{var}\ m : proc = \texttt{pds}$$

where `pds` is a parametrised command, as in the example below.

$$\texttt{valres self:Object} \bullet \texttt{mc} \lhd \texttt{self is C} \rhd (\texttt{mb} \lhd \texttt{self is B} \rhd (\texttt{ma} \lhd \texttt{self is A} \rhd \bot_{po}))$$

Pointer Machine

$$< A : \mathbb{P}Ad,\ V : Ad \mapsto Value,\ S : Ad \leftrightarrow Ad >$$

where

$$Ad \mathrel{\widehat{=}} (seq\ Label) \backslash \{<>\}$$

and

$$Value \mathrel{\widehat{=}} \{T : Type;\ i : name \mid i \in T \bullet i\}$$

Dynamic Types
$dts : Ad \mapsto Type.$

Projections

$$A.p \mathrel{\widehat{=}} \{q : Ad \mid p.q \in A\}$$

$$V.p \mathrel{\widehat{=}} \{q : Ad \mid p.q \in \operatorname{dom} V \bullet q \mapsto V(p.q)\}$$

D.2 Healthiness Conditions

OOs

OO1 $P = P \land \boldsymbol{Object} \in cls$

OO2 $P = P \land \operatorname{dom} sc = cls \setminus \{\boldsymbol{Object}\}$

OO3 $P = P \land \forall C : \operatorname{dom} sc \bullet (C, \boldsymbol{Object}) \in sc^+$

OO4 $P = P \land \operatorname{dom} atts = cls$

OO5 $P = P \land \forall C_1, C_2 : \operatorname{dom} atts \bullet C_1 \neq C_2 \land \operatorname{dom}(atts(C_1)) \cap \operatorname{dom}(atts(C_2)) = \emptyset$

OO6 $P = P \land \operatorname{ran}(\bigcup \operatorname{ran} atts) \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup cls$

OO7 $P = P \land \boldsymbol{Object} \in cls'$

OO8 $P = P \land \operatorname{dom} sc' = cls' \setminus \{\boldsymbol{Object}\}$

OO9 $P = P \land \forall C : \operatorname{dom} sc' \bullet (C, \boldsymbol{Object}) \in sc'^+$

OO10 $P = P \land \operatorname{dom} atts' = cls'$

OO11 $P = P \land \forall C_1, C_2 : \operatorname{dom} atts' \bullet C_1 \neq C_2 \land \operatorname{dom}(atts'(C_1)) \cap \operatorname{dom}(atts'(C_2)) = \emptyset$

OO12 $P = P \land \operatorname{ran}(\bigcup \operatorname{ran} atts') \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup cls'$

OO13 $P = P;\ \mathit{II}_{oo}$

HPs

HP1 $P = P \land \forall\, a_1 : A;\ a_2 : Ad \mid a_2 < a_1 \bullet a_2 \in A$

HP2 $P = P \land \mathrm{dom}\ V = term(A)$

HP3 $P = P \land \bigwedge\{\ '\!x : vars(A) \bullet x =\!'x\}$

where $NPV \cap vars(A) = \emptyset$

and

$\{A, V, S, cls, sc, atts, dts\} \cup M \subseteq NPV$

where : M is the set of method variables.

$vars(X) \mathrel{\widehat{=}} \{x : X \bullet x(1)\}$

$$!x \mathrel{\widehat{=}} \begin{cases} (dts(x), V(x)), & \textit{if } x \in term(A) \\ (dts(x), V.x) & \textit{, otherwise.} \end{cases} \qquad !x' \mathrel{\widehat{=}} \begin{cases} (dts(x'), V'(x)), & \textit{if } x \in term(A') \\ (dts(x'), V'.x) & \textit{, otherwise.} \end{cases}$$

HP4 $P = P \land S \in (A \leftrightarrow A) \land S = S^*$

HP5 $P = P \land fclos_A\ S$

where $fclos_A\ E \mathrel{\widehat{=}} \forall\, x, y : Ad \mid (x, y) \in E \bullet \forall\, a : Ad \mid x.a \in A \land y.a \in A \bullet (x.a, y.a) \in E$

HP6 $P = P \land \forall\, a, b : Ad \bullet (a, b) \in S \land a \in \mathrm{dom}\ \ V \Rightarrow b \in \mathrm{dom}\ V \land V(a) = V(b)$

HP7-12 correspond to **HP1-6** but restricting the final values for A, V and S.

POs

PO1 $P = P \land \mathrm{dom}\ dts = A \land \mathrm{ran}\ dts \subseteq Type$

PO2 $P = P \land \mathrm{dom}\ dts' = A' \land \mathrm{ran}\ dts' \subseteq \{\mathbb{B}, \mathbb{Z}\} \cup cls'$

Other Definitions

$$\mathbf{OOI} \equiv \mathbf{OO1} \circ \mathbf{OO2} \circ \mathbf{OO3} \circ \mathbf{OO4} \circ \mathbf{OO5} \circ \mathbf{OO6}$$

$$\mathbf{OO} \equiv \mathbf{OO1} \circ \mathbf{OO2} \circ \mathbf{OO3} \circ \mathbf{OO4} \circ \mathbf{OO5} \circ \mathbf{OO6} \circ \mathbf{OO7} \circ \mathbf{OO8} \circ \mathbf{OO9} \circ \mathbf{OO10} \circ \mathbf{OO11} \circ \mathbf{OO12}$$

$$\mathit{I\!I}_{oo} \mathrel{\widehat{=}} \mathbf{OOI}(\mathit{I\!I})$$

$$\mathbf{HPI} \equiv \mathbf{HP1} \circ \mathbf{HP2} \circ \mathbf{HP3} \circ \mathbf{HP4} \circ \mathbf{HP5} \circ \mathbf{HP6}$$

$$\mathbf{HP} \equiv \mathbf{HP1} \circ \mathbf{HP2} \circ \mathbf{HP3} \circ \mathbf{HP4} \circ \mathbf{HP5} \circ \mathbf{HP6} \circ \mathbf{HP7} \circ \mathbf{HP8} \circ \mathbf{HP9} \circ \mathbf{HP10} \circ \mathbf{HP11} \circ \mathbf{HP12}$$

$$\mathbf{PO} \equiv \quad \mathbf{PO1} \circ \mathbf{PO2}$$

$$\mathbf{IT} \mathrel{\widehat{=}} \mathbf{OO} \circ \mathbf{HP} \circ \mathbf{PO}$$

$$\bot_{po} \mathrel{\widehat{=}} \mathbf{IT}(\bot)$$

D.3 Declarations

Class Introduction

$$\mathbf{class}\ A\ \mathbf{extends}\ B \mathrel{\widehat{=}} \mathbf{IT}\left(\begin{pmatrix} A \notin \mathit{Type} \land \\ B \in \mathit{cls} \end{pmatrix} \vdash \begin{pmatrix} \mathit{cls}' = \mathit{cls} \cup \{A\} \land \\ \mathit{sc}' = \mathit{sc} \cup \{A \mapsto B\} \land \\ \mathit{atts}' = \mathit{atts} \cup \{A \mapsto \emptyset\} \land \\ w' = w \end{pmatrix}\right)$$

$\mathbf{where}\ w = \mathit{in}\alpha(\mathbf{class}\ A\ \mathbf{extends}\ B) \setminus \{\ \mathit{cls},\ \mathit{sc},\ \mathit{atts}\}$

or

$\mathbf{class}\ A = \mathbf{class}\ A\ \mathbf{extends}\ \textbf{\textit{Object}}$

Attribute Introduction

$$\mathbf{att}\ A\ x : T \mathrel{\widehat{=}} \mathbf{IT}\left(\begin{pmatrix} A \in \mathit{cls} \land \\ x \notin \operatorname{dom} \mathcal{C}(\mathit{atts}, \mathit{cls}) \land \\ T \in \mathit{Type} \end{pmatrix} \vdash \begin{pmatrix} \mathit{atts}' = \mathit{atts} \oplus \{A \mapsto (\mathit{atts}(A) \cup \{x \mapsto T\})\} \land \\ w' = w \end{pmatrix}\right)$$

$\mathbf{where}\ w = \mathit{in}\alpha(\mathbf{att}\ A\ x : T) \setminus \{\mathit{atts}\}$
$\mathbf{and}\ \mathcal{C}(\mathit{amap}, \mathit{cset}) = \bigcup\{N : \mathit{cset} \bullet \mathit{amap}\ N\}$,
amap is an attribute mapping, and *cset* is class set.

Closure of attributes

$$\mathcal{U}(amap, smap, N) = \bigcup amap(\!|\ smap^*(\!|\ \{N\}\ |\!)\ |\!)$$

Method Introduction

$$\textbf{meth}\ A\ m = (pds \bullet p) \mathrel{\widehat{=}} \textbf{IT}\left(\textbf{var}\ m;\ \begin{pmatrix} A \in cls \wedge \\ \forall\, t \in types(pds) \bullet t \in Type \end{pmatrix} \vdash \begin{pmatrix} m' = \mathtt{pds}_e \bullet (\mathtt{p} \lhd \mathtt{self\ is\ A} \rhd \bot_{po}) \wedge \\ w' = w \end{pmatrix}\right)$$

provided $m \notin \alpha(\textbf{meth}\ A\ m = (pds \bullet p))$
where $\mathtt{pds}_e = \mathtt{valres\ self{:}Object;\ pds}$ **and** $w = in\alpha(\textbf{meth}\ A\ m = (pds \bullet p))$

Method Redefinition

$$\textbf{meth}\ A\ m = (pds \bullet p) \mathrel{\widehat{=}} \textbf{IT}\left(\begin{pmatrix} A \in cls \wedge \\ \exists\, \mathtt{q} \bullet m = \mathtt{pds}_e \bullet \mathtt{q} \end{pmatrix} \vdash \left(\exists\, \mathtt{q} \bullet \begin{pmatrix} m = \mathtt{pds}_e \bullet \mathtt{q} \wedge \\ m' = \mathtt{pds}_e \bullet join(A, \mathtt{p}, \mathtt{q}) \wedge \\ w' = w \end{pmatrix}\right)\right)$$

provided $m \in \alpha(\textbf{meth}\ A\ m = (pds \bullet p))$
where $\mathtt{pds}_e = \mathtt{valres\ self{:}Object;\ pds}$, $w = in\alpha(\textbf{meth}\ A\ m = (pds \bullet p)) \setminus \{m\}$
and

$join(A, \mathtt{p}, \bot_{po}) = \mathtt{p} \lhd \mathtt{self\ is\ A} \rhd \bot_{po}$

$$join(A, \mathtt{p}, \mathtt{q}_l \lhd \mathtt{self\ is\ B} \rhd \mathtt{q}_r) = \begin{cases} \mathtt{p} \lhd \mathtt{self\ is\ A} \rhd (\mathtt{q}_l \lhd \mathtt{self\ is\ B} \rhd \mathtt{q}_r), & \text{if } A \prec B \\ \mathtt{q}_l \lhd \mathtt{self\ is\ B} \rhd join(A, \mathtt{p}, \mathtt{q}_r) & \text{, otherwise} \end{cases}$$

$join(A, \mathtt{p}, \mathtt{q}) = \bot_{po}$, *for programs* $\mathtt{q}$ *of every other form*

D.4 Abstractions

Call by Value

$$(\textbf{val}\ v : T \bullet p) \mathrel{\widehat{=}} (\lambda\, w : T \bullet (\textbf{var}\ v : T;\ v := w;\ p;\ \textbf{end}\ v : T))$$

Call by Result

$$(\textbf{res}\ v : T \bullet p) \mathrel{\widehat{=}} (\lambda\, w : \mathcal{N} \bullet (\textbf{var}\ v : T;\ p;\ w := v;\ \textbf{end}\ v : T))$$

Call by Value-Result

$$(\textbf{valres}\ v : T \bullet p) \mathrel{\widehat{=}} (\lambda\, w : \mathcal{N} \bullet (\textbf{var}\ v : T;\ v := w;\ p;\ w := v;\ \textbf{end}\ v : T))$$

Call by Reference

$$(\mathbf{ref}\ v : T \bullet p) \mathrel{\widehat=} (\lambda\, w : \mathcal{N} \bullet (\mathbf{var}\ v : T;\ v :\!\!- w;\ p;\ \mathbf{end}\ v : T))$$

$$(\lambda\, x : \mathcal{N} \bullet p)(y) \mathrel{\widehat=} p\,[y, y', yt, yt'/x, x', xt, xt']$$

D.5 Variables

Declaration

$$\mathbf{var}\ x : T \mathrel{\widehat=} \mathbf{IT}\left(\begin{array}{l}\{T \in Type\}_\perp;\ \mathbf{var}\ xt, x;\ x \notin A \vdash \exists\, v : Value \bullet \\ \qquad \left(\begin{array}{l} xt' = T \land \\ x' \in \mathcal{V}(T) \land \\ A' = A \cup \{x\} \land \\ V' = V \oplus \{x \mapsto v\} \land \\ S' = S \cup \{x \mapsto x\} \land \\ dts' = dts \oplus \{x \mapsto T\} \land \\ w' = w \end{array}\right)\end{array}\right)$$

provided $x \notin in\alpha(\mathbf{var}\ x : T)$
where $w \in in\alpha(\mathbf{var}\ x : T) \setminus \{x, xt, A, V, S, dts\}$

Undeclaration

$$\mathbf{end}\ x : T \mathrel{\widehat=} \mathbf{IT}\left(\left(\left(x \in A \vdash \left(\begin{array}{l} A' = A \setminus \{x\infty \cup \{x\}\} \land \\ V' = (x\infty \cup \{x\}) \lhd V \land \\ S' = (x\infty \cup \{x\}) \lhd S \rhd (x\infty \cup \{x\}) \land \\ dts' = (x\infty \cup \{x\}) \lhd dts \land \\ w' = w \end{array}\right)\right);\ \mathbf{end}\ x, xt\right)\right)$$

provided $x \in in\alpha(\mathbf{end}\ x : T)$
where $w \in in\alpha(\mathbf{end}\ x : T) \setminus \{x, A, dts, V, S\}$

D.6 Expressions

BNF

$$\begin{array}{rcl} e & ::= & v \mid le \mid \mathbf{new}\ N \mid e\ \mathbf{is}\ N \mid (N)e \mid f(e) \mid \mathbf{null} \\ le & ::= & x \mid \mathbf{self} \mid le.x \end{array}$$

Table D.1: BNF for object-oriented expressions.

Well-definedness

Primitive Values

$$\mathcal{D}((\mathbb{B}, v)) \mathrel{\widehat{=}} v \in \mathbb{B}$$
$$\mathcal{D}((\mathbb{Z}, v)) \mathrel{\widehat{=}} v \in \mathbb{Z}$$

Objects

$$\mathcal{D}((T, \mathbf{null})) \mathrel{\widehat{=}} T \in cls$$
$$\mathcal{D}((T, v)) \mathrel{\widehat{=}} T \in cls \land (T, v) \in \mathcal{V}(T)$$

Variables

$$\mathcal{D}(x) \mathrel{\widehat{=}} xt \in Type \land x \in A$$
$$\mathcal{D}(\mathbf{self}) \mathrel{\widehat{=}} \mathbf{self}t \in cls \land \mathbf{self} \in A$$

Attribute Accesses

$$\mathcal{D}(le.x) \mathrel{\widehat{=}} \mathcal{D}(le) \land le_t \in cls \land le_v \neq \mathbf{null} \land x \in \operatorname{dom} le_v$$

Typing

$$\mathcal{D}(\mathbf{new}\ N) \mathrel{\widehat{=}} N \in cls$$
$$\mathcal{D}(e\ \mathbf{is}\ N) \mathrel{\widehat{=}} \mathcal{D}(e) \land N \in cls$$
$$\mathcal{D}((N)e) \mathrel{\widehat{=}} \mathcal{D}(e) \land N \in cls \land e_t \preceq N$$

Remainder

$$\mathcal{D}(x\%y) \mathrel{\widehat{=}} \mathcal{D}(x) \land \mathcal{D}(y) \land x_t = \mathbb{Z} \land y_t = \mathbb{Z} \land y_v \neq 0$$

Object Creation

$$\mathbf{new}\ N \mathrel{\widehat{=}} (N, \{n : name;\ v : Value \mid n \in \operatorname{dom} map \land v \in map(n) \bullet n \mapsto v\})$$

$$\mathbf{where}\ map = \mathcal{U}(atts, cls, N)$$

Type Test

$$e\ \mathbf{is}\ N \mathrel{\widehat{=}} (\mathbb{B}, e_t \preceq N)$$

Type Cast

$$(N)e \mathrel{\widehat{=}} e$$

Attribute Access

$$le.x \mathrel{\widehat{=}} !le.x$$

Equality

$$p =_v q \equiv A.p = A.q \wedge\ !p =!q$$

$$p =_p q \equiv (p, q) \in S$$

D.7 Commands

BNF

$$c ::= le := e \mid \mathit{II} \mid \textbf{var}\ x : T \mid \textbf{end}\ x : T \mid c_1 \lhd e \rhd c_2 \mid c_1;\ c_2 \mid \mu\, X \bullet F(X) \mid le.m(e)$$

Table D.2: BNF for object-oriented commands.

Well-definedness

Assignment to variables
$\mathcal{D}(x := e) \ \widehat{=}\ \mathcal{D}(x) \wedge \mathcal{D}(e) \wedge e_t \preceq xt$

Assignment to attributes
$\mathcal{D}(le.x := e) \ \widehat{=}\ \mathcal{D}(le.x) \wedge \mathcal{D}(e) \wedge e_t \preceq \mathcal{U}(atts, sc, le_t)(x)$

Conditional
$\mathcal{D}(P \lhd e \rhd Q) \ \widehat{=}\ \mathcal{D}(e) \wedge e_t = \mathbb{B}$

Method call
$\mathcal{D}(le.m(e)) \ \widehat{=}\ \mathcal{D}(le) \wedge le_v \neq \textbf{null} \wedge \mathit{compatible}(le, m) \wedge e_t \preceq T$

provided $\exists \mathtt{p} \bullet m = (\mathtt{val\ x{:}T} \bullet \mathtt{p})$,
where $\mathit{compatible}(le, m) \equiv \exists \mathtt{pds}, \mathtt{p} \bullet m = (\mathtt{pds} \bullet \mathtt{p}) \wedge le_t \in \mathit{scan}(\mathtt{p})$
with
$\mathit{scan}(\bot_{po}) = \{\}$
$\mathit{scan}(\mathtt{p}_l \lhd \mathtt{self\ is\ A} \rhd \mathtt{p}_r) = \{B : \mathit{cls} \mid B \preceq A\} \cup \mathit{scan}(\mathtt{p}_r)$

$\mathcal{D}(le.m(y)) \ \widehat{=}\ \mathcal{D}(le) \wedge le_v \neq \textbf{null} \wedge \mathit{compatible}(le, m) \wedge \mathit{frame}(le) \cap \mathit{frame}(y) = \emptyset \wedge T \preceq y_t$
provided $\exists \mathtt{p} \bullet m = (\mathtt{res\ x{:}T} \bullet \mathtt{p})$

$\mathcal{D}(le.m(z)) \ \widehat{=}\ \mathcal{D}(le) \wedge le_v \neq \textbf{null} \wedge \mathit{compatible}(le, m) \wedge \mathit{frame}(le) \cap \mathit{frame}(y) = \emptyset \wedge T = z_t$
provided $\exists \mathtt{p} \bullet m = (\mathtt{valres\ x{:}T} \bullet \mathtt{p})$

$\mathcal{D}(le.m(e)) \ \widehat{=}\ \mathcal{D}(le) \wedge le_v \neq \textbf{null} \wedge \mathit{compatible}(le, m) \wedge e_t \preceq T$
provided $\exists \mathtt{p} \bullet m = (\mathtt{ref\ x{:}T} \bullet \mathtt{p})$

Assignments of New Objects

$$
\begin{array}{l}
x :=_r \textbf{new}\ N \mathrel{\widehat{=}} \\
\quad \textbf{IT}\left(\begin{array}{l} \mathcal{D}(x := \textbf{new}\ N) \vdash \left(\begin{array}{l} A' = (A \setminus x{\infty}) \cup \{n : name \mid n \in \operatorname{dom} map \bullet x.n\} \wedge \\ V' = ((x{\infty} \cup \{x\}) \lhd V) \oplus \{n : name \mid n \in \operatorname{dom} map \bullet x.n \mapsto map(n)\} \wedge \\ S' = (x{\infty} \cup \{x\}) \lhd S \rhd (x{\infty} \cup \{x\}) \wedge \\ dts' = ((x{\infty} \cup \{x\}) \lhd dts) \oplus (\{x \mapsto N\} \cup \{x.f \mapsto aclos(f) \mid f \in \operatorname{dom} map\}) \wedge \\ w' = w \end{array}\right) \end{array}\right) \\
\qquad \textbf{where}\ (N, map) = \textbf{new}\ N \\
\qquad \textbf{with}\ aclos = \mathcal{U}(atts, cls, N) \\
\qquad \textbf{and}\ w \in in\alpha(x :=_r \textbf{new}\ N) \setminus \{A, V, S, dts\}
\end{array}
$$

Value Assignments of Variables or Attributes

$$
\begin{array}{l}
x := e \mathrel{\widehat{=}} \\
\quad \textbf{IT}\left(\begin{array}{l} \mathcal{D}(x := e) \vdash \left(\begin{array}{l} A' = A \setminus ext_S(x) \wedge \\ V' = ((ext_S(x) \cup share_S(x)) \lhd V) \oplus \{a : (share_S(x) \setminus ext_S(x)) \bullet a \mapsto e_v\} \wedge \\ S' = ext_S(x) \lhd S \rhd ext_S(x) \wedge \\ dts' = ((ext_S(x) \cup share_S(x)) \lhd dts) \oplus \{a : (share_S(x) \setminus ext_S(x)) \bullet a \mapsto e_t\} \end{array}\right) \end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
x := y \mathrel{\widehat{=}} \\
\quad \textbf{IT}\left(\begin{array}{l} \mathcal{D}(x := y) \\ \vdash \\ \left(\begin{array}{l} A' = (A \setminus x{\infty}) \cup \{a : Ad \mid y.a \in A \bullet x.a\} \wedge \\ dts' = (x{\infty} \lhd dts) \oplus (\{a : \operatorname{dom} V;\ fa : seq\ Label \mid a = y.fa \bullet x.fa \mapsto dts(y.fa)\} \cup \{x \mapsto dts(y)\}) \wedge \\ V' = (x{\infty} \lhd V) \oplus \{a : \operatorname{dom} V;\ fa : seq\ Label \mid a = y.fa \bullet x.fa \mapsto V(y.fa)\} \wedge \\ S' = ((x{\infty} \cup \{x\}) \lhd S \rhd (x{\infty} \cup \{x\})) \end{array}\right) \end{array}\right)
\end{array}
$$

Pointer Assignments of Variables or Attributes

$$
\begin{array}{l}
x :- y \mathrel{\widehat{=}} \\
\quad \textbf{IT}\left(\begin{array}{l} \mathcal{D}(x :- e) \\ \vdash \\ \left(\begin{array}{l} A' = (A \setminus x{\infty}) \cup \{a : Ad \mid y.a \in A \bullet x.a\} \wedge \\ V' = (x{\infty} \lhd V) \oplus \{a : \operatorname{dom} V;\ fa : seq\ Label \mid a = y.fa \bullet x.fa \mapsto V(y.fa)\} \wedge \\ S' = (((x{\infty} \cup \{x\}) \lhd S \rhd (x{\infty} \cup \{x\})) \cup \{x \mapsto y\} \cup \{a : Ad \mid y.a \in A \bullet x.a \mapsto y.a\})^* \wedge \\ dts' = (x{\infty} \lhd dts) \oplus (\{a : \operatorname{dom} V;\ fa : seq\ Label \mid a = y.fa \bullet x.fa \mapsto dts(y.fa)\} \cup \{x \mapsto dts(y)\}) \end{array}\right) \end{array}\right)
\end{array}
$$

Conditional

$$P \lhd e \rhd Q \mathrel{\widehat{=}} \textbf{IT}(\mathcal{D}(P \lhd e \rhd Q) \wedge ((e_v \wedge P) \vee (\neg e_v \wedge Q)))$$

Simple Recursion

$$\textbf{meth}\ A\ m = \mu\, X \bullet \big(\ pds \bullet F(X)\ \big)$$

Mutual Recursion

$$\textbf{meth}\ A\ m, B\ n = \mu\, X, Y \bullet \big(\ pds_m \bullet F(X, Y), pds_n \bullet G(X, Y)\ \big)$$

Method Call

$$le.m(args) \mathrel{\widehat{=}} \mathbf{IT}(\{\mathcal{D}(le.m(args))\}_\perp;\ (pds_e \bullet p)(le, args))$$

where $m = \mathtt{pds}_e \bullet \mathtt{p}$

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.

[AL97] Martín Abadi and K. Rustan M. Leino. A Logic of Object-Oriented Programs. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer, 1997.

[AL03] Martín Abadi and K. Rustan M. Leino. A Logic of Object-Oriented Programs. In Nachum Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 11–41. Springer, 2003.

[Ame90] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *REX Workshop*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer, 1990.

[Bac87] R. J. R. Back. Procedural Abstraction in the Refinement Calculus. Technical report, Department of Computer Science, Åbo, Finland, 1987. Ser. A No. 55.

[BCC+03] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS '03)*, volume 80 of *Electronic Notes in Theoretical Computer Science*, pages 73–89. Elsevier, June 2003.

[BCD+06] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 364 – 387. Springer Berlin / Heidelberg, 2006.

[BDF+04] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004. Special issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs.

[BF98] Jean-Michel Bruel and Robert B. France. Transforming UML models to Formal Specifications. In Pierre-Alain Muller and Jean Bézivin, editors, *Proc. International Conference on the Unified Modelling Language (UML): Beyond the Notation*, number 1618. Springer-Verlag, 1998.

[BJR98] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1st edition, September 1998.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer Berlin / Heidelberg, 2005.

[BM06] F. Bannwart and P. Müller. Changing programs correctly: Refactoring with specifications. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Formal Methods (FM)*, volume 4085 of *Lecture Notes in Computer Science*, pages 492–507. Springer-Verlag, 2006.

[BMvW00] Ralph-Johan Back, Leonid Mikhajlov, and Joakim von Wright. Formal Semantics of Inheritance and Object Substitutability. Technical Report TUCS-TR-337, 27, 2000.

[Bor07] Borland Software Corporation. Borland: JBuilder, 2007. At http://www.borland.com/br/products/jbuilder/. Last accessed in 06/03/2007.

[BRL03] Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. Java Applet Correctness: A Developer-Oriented Approach. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer, 2003.

[Bro86] Stephen D. Brookes. A fully abstract semantics and a proof system for an ALGOL-like language with sharing. In *Proceedings of the international conference on Mathematical foundations of programming semantics*, pages 59–100, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[BS00] P. H. M. Borba and A. C. A. Sampaio. Basic Laws of ROOL: an object-oriented language. In *3rd Workshop on Formal Methods*, pages 33–44, Brazil, 2000.

[BSC03] P. H. M. Borba, A. C. A. Sampaio, and M. L. Cornélio. A Refinement Algebra for Object-Oriented Programming. In Luca Cardelli, editor, *ECOOP*, volume 2743 of *Lecture Notes in Computer Science*, pages 457–482. Springer, 2003.

[BSCC04] P. H. M. Borba, A. C. A. Sampaio, A. L. C. Cavalcanti, and M. L. Cornélio. Algebraic Reasoning for Object-Oriented Programming. *Science of Computer Programming*, 52:53–100, 2004.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998. Graduate Texts in Computer Science.

[CHW06] A. L. C. Cavalcanti, W. Harwood, and J. C. P. Woodcock. Pointers and Records in the Unifying Theories of Programming. In S. Dunne and B. Stoddart, editors, *Unifying Theories of Programming*, volume 4010 of *Lecture Notes in Computer Science*, pages 200 – 216. Springer-Verlag, 2006.

[CN00] A. L. C. Cavalcanti and D. A. Naumann. A Weakest Precondition Semantics for Refinement of Object-oriented Programs. *IEEE Transactions on Software Engineering*, 26(8):713–728, 2000.

[Coo89] William R. Cook. A Proposal for Making Eiffel Type-Safe. *The Computer Journal*, 32(4):305–311, 1989.

[Cor04] M. L. Cornélio. *Refactorings as Formal Refinements.* PhD thesis, Centre de Informatic - Federal University of Pernambuco, 2004.

[CSW05] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Unifying Classes and Processes. *Software and System Modelling*, 4(3):277–296, 2005.

[Dan02] Daniel Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.

[DE98] S. Drossopoulou and S. Eisenbach. *Towards an Operational Semantics and Proof of Type Soundness for Java.* Springer-Verlag, March 1998.

[DS95] Jim Davies and Steve Schneider. A brief history of Timed CSP. In *MFPS '92: Selected papers of the meeting on Mathematical foundations of programming semantics*, pages 243–271, Amsterdam, The Netherlands, The Netherlands, 1995. Elsevier Science Publishers B. V.

[DW06] Donovan Wells. Extreme Programming: A Gentle Introduction., 2006. At http://www.extremeprogramming.org/. Last accessed in 15/01/2007.

[EFLR99] A. Evans, R. France, K. Lano, and B. Rumpe. The UML as a Formal Modeling Notation. In J. Bézivin and P. Muller, editors, *The Unified Modling Language, UML'98 – Beyond the Notation*, volume 1618 of *Lecture Notes in Computer Science*, pages 336–348. Springer-Verlag, 1999.

[EPG+06] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2006.

[Eva98] A. Evans. Reasoning with UML Diagrams. In *Workshop on Industrial Strength Formal Methods, WIFT'98*. IEEE Press, 1998.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Reading, MA, USA, 1999.

[FEL97] Robert France, Andy Evans, and Kevin Lano. The UML as a Formal Modeling Notation. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Proceedings OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, pages 75–81. Technische Universität München, TUM-I9737, 1997.

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an Annotation Assistant for ESC/Java. In José Nuno Oliveira and Pamela Zave, editors, *FME*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, 2001.

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

[FOW01] C. Fischer, E. R. Olderog, and H. Wehrheim. A CSP view on UML-RT structure diagrams. In Heinrich Hussmann, editor, *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001, Proceedings*, volume 2029 of *LNCS*, pages 91–108. Springer, 2001.

[GH93] John V. Guttag and James J. Horning. *Larch: languages and tools for formal specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, January 1995.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification*. Addison-Wesley, 2nd edition, June 2000.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, 1993.

[HCW07] W. Harwood, A. L. C. Cavalcanti, and J. C. P. Woodcock. A Model of Pointers for the Unifying Theories of Programming. Technical report, Citrix Systems (R & D) Ltd and University of York, 2007.

[Heh04] E.C.R. Hehner. *A Practical Theory of Programming, the second edition*. Springer-Verlag, New York, 2004.

[HH98] C. A. R. Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[HHJ^{+}87] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Commun. ACM*, 30(8):672–686, 1987.

[HHJ^{+}92] C. A. R. Hoare, I. J. Hayes, H. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorenson, J. M. Spivey, and B. A. Sufrin. Laws of Programming. In M. Broy, editor, *Programming and Mathematical Method*, pages 95–122. Springer, Berlin, Heidelberg, 1992.

[HLL05] Jifeng He, Xiaoshan Li, and Zhiming Liu. A Refinement Calculus for Object Systems. Technical report 322, UNU-IIST, P.O.Box 3058, Macau, 2005.

[HLL06] Jifeng He, Xiaoshan Li, and Zhiming Liu. rCOS: A refinement calculus of object systems. *Theor. Comput. Sci.*, 365(1-2):109–142, 2006.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[Hui01] M. Huisman. *Reasoning about Java Programs in Higher Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[Jon90] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.

[Kas05] Ioannis T. Kassios. Decoupling in Object Orientation. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 43–58. Springer, 2005.

[KW98] A. Kleppe and J. Warmer. *The Object Constraint Language : Precise Modeling with UML*. Object Technology Series. Addison-Wesley, 1st edition, October 1998.

[LB98] Kevin Lano and Juan Bicarregui. Semantics and Transformations for UML Models. In Jean Bézivin and Pierre-Alain Muller, editors, *UML*, volume 1618 of *Lecture Notes in Computer Science*, pages 107–119. Springer, 1998.

[LBE00] Kevin Lano, Juan Bicarregui, and Andy Evans. Structured Axiomatic Semantics for UML Models. In *Rigorous Object-Oriented Methods*, Workshops in Computing. BCS, 2000.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[Lei98] K. Rustan M. Leino. Recursive Object Types in a Logic of Object-Oriented Programs. *Nordic Journal of Computing*, 5(4):330–360, Winter 1998.

[LLM06a] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 2006. to appear.

[LLM06b] Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. Technical Report 06-14a, Department of Computer Science, Iowa State University, Ames, Iowa, Aug 2006.

[Ltd89] International Computers Ltd. Proofpower, 1989. At http://www.lemma-one.com/ProofPower/index/index.html. Last accessed in 19/06/2007.

[LW94] B. H. Liskov and J. M. Wing. A Behavioural Notion of Subtyping. *ACM Trans. on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[MD98] B. Mahony and J.S. Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104, Kyoto, Japan, April 1998. IEEE Computer Society Press.

[MD00] Brendan P. Mahony and Jin Song Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, 2000.

[Mey92] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall International, second edition edition, 1997.

[Mic07a] Microsoft Corporation. The C# Language, 2007. At http://msdn2.microsoft.com/en-us/vcsharp/aa336809.aspx. Last accessed in 16/01/2007.

[Mic07b] Microsoft Corporation. Visual Studio - Developer Center, 2007. At http://msdn2.microsoft.com/en-us/vstudio/default.aspx. Last accessed in 06/03/2007.

[Mor94] C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.

[MPH97] P. Müller and A. Poetzsch-Heffter. Formal specification techniques for object-oriented programs. In M. Jarke, K. Pasedach, and K. Pohl, editors, *Informatik 97: Informatik als Innovationsmotor*, Informatik Aktuell, pages 602–611. Springer-Verlag, 1997.

[MS76] R. Milne and C. Strachey. *A theory of programming language semantics*. Chapman and Hall, 1976.

[MS97] Anna Mikhajlova and Emil Sekerinski. Class Refinement and Interface Refinement in Object-Oriented Programs. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313, pages 82–101. Springer-Verlag, 1997.

[Nau95] David A. Naumann. Predicate transformers and higher-order programs. *Theor. Comput. Sci.*, 150(1):111–159, 1995.

[NE02] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 229–239, New York, NY, USA, 2002. ACM Press.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[NvO98] Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*. ACM Press, New York, 1998.

[OCW06] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. Unifying Theories in ProofPower-Z. In S. Dunne and B. Stoddart, editors, *UTP2006: the first international symposium of unifying theories of programming*, volume 4010 of *Lecture Notes in Computer Science*, pages 123–140. Springer-Verlag, 2006. Springer-Verlag.

[Oli05] M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, Department of Computer Science - University of York, UK, 2005. YCST-2006-02.

[OMG97] OMG. Unified Modeling Language, 1997. Object Management Group. Available at: http://www.omg.org/uml.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[ORS92] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994. LNCS 828.

[PdB03] Cees Pierik and Frank S. de Boer. A Syntax-Directed Hoare Logic for Object-Oriented Programming Concepts. In Elie Najm, Uwe Nestmann, and Perdita Stevens, editors, *FMOODS*, volume 2884 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2003.

[PHM98] Arnd Poetzsch-Heffter and Peter Müller. Logical foundations for typed object-oriented languages. In David Gries and Willem P. de Roever, editors, *PROCOMET*, volume 125 of *IFIP Conference Proceedings*, pages 404–423. Chapman & Hall, 1998.

[PHM99] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential java. In S. Doaitse Swierstra, editor, *ESOP*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 1999.

[Plo81] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, September, 1981.

[PMP01] Zsigmond Pap, István Majzik, and András Pataricza. Checking general safety criteria on uml statecharts. In Udo Voges, editor, *SAFECOMP*, volume 2187 of *Lecture Notes in Computer Science*, pages 46–55. Springer, 2001.

[PO04] Richard F. Paige and Jonathan S. Ostroff. ERC – An object-oriented refinement calculus for Eiffel. *Form. Asp. Comput.*, 16(1):51–79, 2004.

[QDC03] S. C. Qin, J. S. Dong, and W. N. Chin. A Semantic Foundation of TCOZ in Unifying Theory of Programming. In *FM'03*, Lecture Notes in Computer Science, pages 321–340, Pisa, Italy, September 2003. Springer-Verlag.

[Qia99] Zhenyu Qian. A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subrountines. In *Formal Syntax and Semantics of Java*, pages 271–312, 1999.

[Ral00] Ralph-Johan Back and Anna Mikhajlova and Joakim von Wright. Class Refinement as Semantics of Correct Object Substitutability. *Formal Asp. Comput.*, 12(1):18–40, 2000.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[RG98] Mark Richters and Martin Gogolla. On formalizing the UML Object Constraint Language OCL. In Tok Wang Ling, Sudha Ram, and Mong-Li Lee, editors, *ER*, volume 1507 of *Lecture Notes in Computer Science*, pages 449–464. Springer, 1998.

[RG02] M. Richters and M. Gogolla. OCL: Syntax, semantics, and tools. In Tony Clark and Jos Warmer, editors, *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, pages 42–68. Springer, 2002.

[Rob99] Donald B. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.

[Sch86] D. A. Schmdit. *Denotational Semantics. A Methodology for Language Development*. Allyn and Bacon,Inc, 1986.

[SCS06] Thiago Santos, Ana Cavalcanti, and Augusto Sampaio. Object-Orientation in the UTP. In S. Dunne and B. Stoddart, editors, *UTP 2006: First International Symposium on Unifying Theories of Programming*, volume 4010 of *LNCS*, pages 20–38. Springer-Verlag, 2006.

[SH02] Adnan Sherif and Jifeng He. Towards a Time Model for *Circus*. In Chris George and Huaikou Miao, editors, *ICFEM*, volume 2495 of *Lecture Notes in Computer Science*, pages 613–624. Springer, 2002.

[Smi00] Graeme Smith. *The Object-Z Specification Language*. Advances in Formal Methods Series. Kluwer Academic Publishers, 2000.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 2nd edition, 1992.

[Str85] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1985.

[Sym99] Don Syme. Proving Java Type Soundness. In *Formal Syntax and Semantics of Java*, pages 83–118, 1999.

[TB01] Lance Tokuda and Don S. Batory. Evolving object-oriented designs with refactorings. *Autom. Softw. Eng.*, 8(1):89–120, 2001.

[The07] The Eclipse Foundation. Eclipse.org home, 2007. At http://www.eclipse.org/. Last accessed in 06/03/2007.

[Utt92] Mark Utting. *An Object-Oriented Refinement Calculus with Modular Reasoning*. PhD thesis, Kensington, Australia, 1992.

[vdBJ01] Joachim van den Berg and Bart Jacobs. The LOOP Compiler for Java and JML. In Tiziana Margaria and Wang Yi, editors, *TACAS*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer, 2001.

[vO00] David von Oheimb. Axiomatic semantics for Java$^{\ell ight}$ in Isabelle/HOL. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Technical Report 269, 5/2000, Fernuniversität Hagen, Fernuniversität Hagen, 2000.

[vON99] David von Oheimb and Tobias Nipkow. Machine-Checking the Java Specification: Proving Type-Safety. In *Formal Syntax and Semantics of Java*, pages 119–156, 1999.

[WC02] J. C. P. Woodcock and A. L. C. Cavalcanti. The Semantics of *Circus*. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer-Verlag, 2002.

[WC04] J. C. P. Woodcock and A. L. C. Cavalcanti. A Tutorial Introduction to Designs in Unifying Theories of Programming. In E. A. Boiten, J. Derrick, and G. Smith, editors, *IFM 2004: Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, pages 40–66. Springer-Verlag, 2004. Invited tutorial.

[WD96] J. C. P. Woodcock and J. Davies. *Using Z-Specification, Refinement, and Proof*. Prentice-Hall, 1996.