# UML and its Meaning

P. H. Schmitt

Winter 2002/2003

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 History

The Unified Modeling Language (UML) is a language for visualizing, specifying, constructing and documenting object-oriented software systems. It has been widely accepted as a standard for modeling software systems and is supported by a great number of CASE tools (Computer Aided Software Engineering tools).

The Unified Modeling Language (UML), version UML 1.1, was adopted as a standard of the Object Management Group (OMG) November 14, 1997. Work on UML was initialized by Grady Booch, James Rumbaugh and Ivar Jacobson by the mid 1990s. The initial focus was to combine and unify the Booch method with OMT, the method developed by James Rumbaugh, and OOSE, Ivar Jacobson's method. The UML project was gradually joined by other researchers till a core team of about twenty finally hold responsible for UML 1.1. Since 1997 the maintenance of the UML standard was taken over by the OMG Revision Task Force (RTF). The current version as of this writing is UML 1.3.

The standard document on UML is [OMG, 2000b]. A comprehensive account of UML may be found in the books authored by the three pioneers [Rumbaugh *et al.*, 1998, Rumbaugh *et al.*, 1999a] accompanied by the description of a process model on the basis of UML in [Rumbaugh *et al.*, 1999b]. The fast road to learn UML is provided by [Fowler & Scott, 1997]. The Object Constraint Language (OCL) is part of UML [OMG, 2000a]. The only available introduction at the time is [Warmer & Kleppe, 1999].

## 1.2 Set Theoretical Notation

We give a short review of the pieces of set theoretical notation used in the following.

### Basic concepts

1. A *set* is the combination of certain well-distiguished objects taken from our visual or mental experience into one entity. The objects are called the elements of the set.

Let $M$ denote a set, and $m$ an object. The fact that $m$ is an element of $M$ is denoted by $m \in M$.

2. A *function* from a set $M_1$ in a set $M_2$ associates with an element $m_1 \in M_1$ a unique element $m_2 \in M_2$. If $f$ is used to denote a function this association is symbolically expressed as $f(m_1) = m_2$. The element $m_1$ is called the argument and $m_2$ the value of the function application $f(m_1)$.

3. A *relation* describes properties of elements, pairs of elements or in general $n$-tupels of elements. If $r$ denotes a unary relation, and $a$ is an object, then $r(a)$ denotes the fact that the relation $a$ is true of the object $a$. For a binary relation $r_2$ and objects $a_1$, $a_2$ the symbolic notation $r_2(a_1, a_1)$ expresses that the relation $r_2$ is true of the pairs $a_1, a_2$.

We consider these three notions as basic. Therefore we do not attempt to define them in terms of simpler or other concepts. Basic in the same sense are also

1. The element relation, $m \in M$, $m$ is an element of the set $M$

2. Function application, $f(a_1, \ldots, a_n)$, which denotes the value of the $n$-ary function $f$ applied to the arguments $(a_1, \ldots, a_n)$.

3. Relation application, $r(a_1, \ldots, a_n)$, which yields the value *true*, when the $n$-ary relation $r$ holds of the tupel $(a_1, \ldots, a_n)$ and the value *false* otherwise.

It is, of course, possible to construct mutual dependencies between these three basic concepts. This is not necessary at the level of presentation of the next 3 chapters.

The explanation given for sets is an attempt to translate Georg Cantor's explanation into English. Georg Cantor is the founding father of modern set theory through his bold publication [Cantor, 1895].

In Cantor's theory sets could be finite or infinite. In fact, it was the inclusion and systematic treatment of infinite sets that made his work so provocatice to the mathematicians of his time.

**Definition 1 (Subset)**
*A set $N$ is called a subset of set $M$ if every element of $N$ is also an element of $M$. In this situation $M$ is also called a superset of $N$ in this case. We write $N \subseteq M$ in this case.*

For small finite sets $M$ we may define $M$ by enumerating all its elements $M = \{a_1, \ldots a_n\}$. The empty set is denoted by $\emptyset$. We use special reserved symbols to denote frequently occuring sets: $\mathbb{N}$, natural numbers, $\mathbb{Z}$, integers, $\mathbb{Q}$, rational numbers, $\mathbb{R}$, real numbers. In most cases, a set $M$ will be defined by singling out elements from a superset $N$ that satisfy a certain property $\phi$, in symbols $M = \{x \in N \mid \phi\}$.

There is still another common way do define particular sets: it starts by singling out elements from a superset, but these elements are not collected themselves into a new set. From these elements, say $a_1, \ldots, a_k$ new ones are constructed, which we may denote by $t(a_1, \ldots, a_k)$, and these then form the elements of the new set. We may compress this explanation into the formal scheme $M = \{t(x_1, \ldots, x_n) \mid \phi(x_1, \ldots, x_n)\}$.

**Examples for defining sets**

1. $\{x \in \mathbb{N} \mid x \text{ is prime}\}$
2. $\{x^2 \mid x \in \mathbb{N} \text{ and } x \text{ is prime}\}$
3. $\{x_1^2 + x_2^2 + x_3^2 + x_4^2 \mid x_i \in \mathbb{Z}\}$

**Definition 2 (Functions)**

1. *A $n$-ary function $f : M_1 \to M_2$ is called* total *if for every $n$-tupel $(a_1, \ldots, a_n)$ of elements from $M_1$ the function value $f(a_1, \ldots, a_n)$ is defined. If this is not the case then $f$ is called* partial.
2. *The set $\{f(a_1, \ldots, a_n) \mid a_1, \ldots, a_n \in M_1\}$ is called the* range *of $f$.*
3. *If $f : M_1 \to M_2$ is a unary function the set of all elements $m \in M_1$ such that $f(m)$ is definied, is called the domain of $f$.*

**Definition 3**
*Let $r$ be a relation.*

1. *The set* $\{a \mid \text{there exists } b \text{ with } r(a,b)\}$ *is called the domain of* $r$
2. *The set* $\{b \mid \text{there exists } a \text{ with } r(a,b)\}$ *is called the range of* $r$

**Definition 4**
*Let* $A$, $B$ *be sets.*

1. *The intersection* $A \cap B$ *is the set of elements occuring both in* $A$ *and* $B$, *i.e.* $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$
2. *The union* $A \cup B$ *is the set of elements occuring either in* $A$ *or* $B$, *i.e.* $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$
3. $A$ *and* $B$ *are called disjoint, if they have no elements in common, i.e.* $A \cap B = \emptyset$.

**Definition 5**
*Let* $A$ *be a set.*

1. *The set of all subsets of* $A$ *is denoted by* $Set(A)$, *i.e.*
$$Set(A) = \{B \mid B \subseteq A\}$$
$Set(A)$ *is also called the* power set *of* $A$.
2. *The set of all finite subset of* $A$ *is denoted by* $Set_\omega(A)$, *i.e.*
$$Set_\omega(A) = \{B \mid B \subseteq A \text{ and } B \text{ is finite}\}$$
3. *For each natural number* $n \in \mathbb{N}$ *the set of all subsets with exactly (at most)* $n$ *elements is denoted by* $Set_n(A)$ *(*$Set_{\leq n}(A)$*).*

**Definition 6**
*A* bag *is a collection where multiple occurences of objects are possible. Bags are sometimes also called multisets.*

*If* $B$ *is a bag and* $e$ *an arbitrary object the function* $count_B(e)$ *denotes the number of occurences of* $e$ *in* $B$.

While a set abstracts from the order and multiplicity of its objects a bag only abstracts from their order.

Note that $count_B(e) > 0$ is equivalent to $e \in B$.

**Examples** $\{a, b, a, c, b\}$ and $\{a, b, c\}$ are the same set, but they are different as bags. $\{a, b, a, c, b\}$ and $\{c, b, a, b, a\}$ are identical bags.

# Chapter 2

# UML Class diagrams

In UML class diagrams are used to model the static design view of a system. They are the most commonly used diagram type. We will look at the elements that make up class diagrams in detail one after the other. In each case we will start with the graphical representation of the item in question, then give its semantics and wrap up with comments.

Before we start on this tour we should explain our understanding of the term *semantics*. So far all approaches to provide UML with a rigorous meaning followed the same line of attack: they translated UML diagrams into a formalism with well-understood semantics. Here is a non-exhaustive list of papers on translations of UML into

1. the CASL-LTL language, an extension of CASL, [Reggio *et al.*, 2000].
2. **Z**, [France, 1999] .
3. Object-**Z**, [Kim & Carrington, 1999] .
4. the logical language of PVS, [Krishnan, 2000].
5. MSM (Mathematical System Model), [Breu *et al.*, 1998].
6. BOTL, [Distefano *et al.*, 2000].
7. LSL, the Larch Shared Language, [Hamie *et al.*, 1998, Hamie, 1998].
8. EER (extended entity relationship), [Gogolla & Richters, 1998] ,
9. Maude in [Álvarez & Alemán, 2000] .
10. COQ in [Russo, 2001] (???).

The main motivation for these translations in most cases was the extra benefit that the translated models could be used as input to some reasoning or analysis tool.

We will take a different avenue and describe the meaning of UML class diagrams by using only the simplest notions from set theory. This approach is modeled after the long standing practise in mathematics, where the proverbial mathematical rigour is obtained without the use of a formalized language or logic, natural language plus some notational conventions suffice. The main goal is to keep things as simple as possible. A comparable stand is taken in

[Cohen, 1998]. Also in [France, 1999], even though the formal language **Z** is used, the emphasis is

> ... to develop precise semantics for UML notations, expressed in a form that is widely understood (e.g., natural language), and that supports rigorous analyses of the models.

It is our intention that the semantics described in this and the next chapter will serve as an easily accessible common basis for translations of the kind mentioned above. No knowledge of a particular formal language will be presupposed. The informal but rigorous descriptions can then be cast into the formal language of one's own choice. There are also no principle obstacles to formalize them in the uniform framework proposed in [Clark *et al.*, 2000].

This being said, let's get down to business. What is the meaning of a UML model? This is made sufficiently clear in [Rumbaugh *et al.*, 1998, pages 59–60]

> One purpose of a model is to describe the possible states of a system and their behavior. A model is a statement of potentiality, of the possible collections of objects that might exist and the possible behavior history that the objects might undergo. The static view defines and constraints the possible configurations of values that an executing system may assume. The dynamic view defines the ways in which an executing system may pass from one configuration to another. .... A particular static configuration of a system at one instant is called a snapshot.

We will concentrate here on the static view only. A *snapshot* is also called a configuration, or the static part of a system state. To get into the right mood let us quote another paragraph from [Rumbaugh *et al.*, 1998]

> The static view defines the set of objects, values, and links that can exist in a single snapshot. In principle, any combination of objects and links that is consistent with a static view is a possible configuration of the model. This does not mean that every possible snapshot can or will occur.

In the following we will thus describe for each model element what their snapshots look like and what are its consistency requirements. As a rule we will at first only consider the most important features of a model elment and return later to add more advanced ones. Taken together the individual descriptions will add up to a snapshot of a whole class diagram.

We work under the assumption that syntactical correctness of class diagrams has been check. Therefore questions about meta-modeling will not be considered.

## 2.1 Classes and Attributes

### 2.1.1 Example

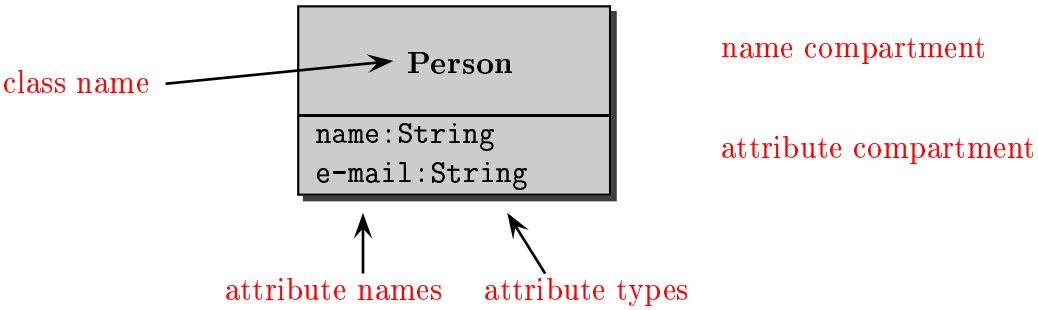



Figure 2.1: Class Person

### 2.1.2 Semantics

In any snapshot a class is interpreted as a set of elements. There is no requirement that the set be non-empty.

Attributes are functions from their class to their type. These functions may be partial, i.e. they need not return a value for every element in their class. The reason for partiality is, that we also want to cover situations where attributes have not yet been initialized.

We will use the same class and attribute names also for their semantic counterparts in a snapshot. In every snapshot of the diagram shown in Figure 2.1 *Person* will be an arbitrary set. In the same snapshot there are also a set *String* and functions *name* and *e-mail* from the set *Person* to the set *String*.

### 2.1.3 Comments

The type of an attribute, as described by the UML metamodel, is a role name of an association between the metaclass *Attribute* and the metaclass *Classifier*. It is not to be confused with the metaclass *DataType*. Any class, interface or data type may occur as the type of an attribute.

UML allows multiplicities on attributes. Though by far the most common multiplicity and the default is 1. Notice, even attributes of default multiplicity 1 will in our framework be considered as partial functions, since they have no value before initialization.

We call attributes with multiplicities different from 1 set-valued attributes. A set-valued attribute in class $C$ of type $T$ will in any snapshot be interpreted as a function from $C$ to the power set of $T$, i.e. the set of all subsets of $T$. Restrictions on the cardinality of the set of values in any snapshot will we added as a consistency requirement. An example of an attribute with non-default multiplicity is the *authors* attribute in Figure 2.2.

Even the exceptional case of multiplicity 0 is explicitely included. This seems to have not been widely accepted. In [Fowler & Scott, 1997] e.g. this possibility is not even mentioned. We would interpret an attribute with multiplicity 0 as a set valued function, whose value is always the empty set.

In most cases an attribute will be a total function. Partiality is neccessary to allow for the situation that for newly created elements the value of an attribute has not been initialized yet, see [Rumbaugh *et al.*, 1998, pages 167 and 303].

As stated above, we try to get along in this chapter with minimal formality.

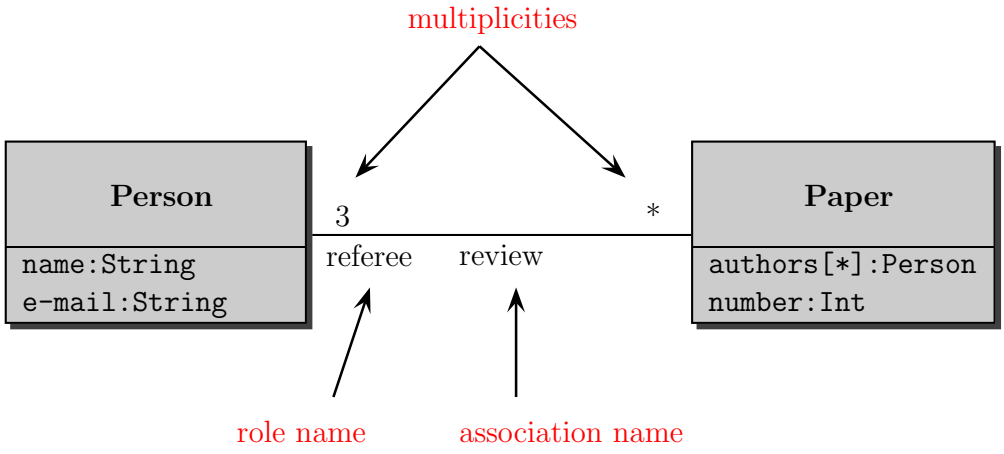



Figure 2.2: The review association

## 2.2 Associations

### 2.2.1 Example

### 2.2.2 Semantics

An association $r$ between classes $C1$ and $C2$ is interpreted in a snapshot as a relation between the sets $C1$ and $C2$. For every pair of elements $c_1$ from $C1$ and $c_2$ from $C2$ we use $r(c_1, c_2)$ to denote that the relation $r$ holds for $c_1$ and $c_2$. The best way to think of instances of an association is as pairs of objects.

A multiplicity is interpreted as a subset of the natural numbers.

| | | |
|---|---|---|
| 0..1 | *is* | $\{0, 1\}$ |
| 0..∗ | *is* | the set of all natural numbers |
| ∗ | *is* | same as previous line |
| 1..3 | *is* | the set of all numbers between 1 and 3 including end points |
| 7 | *is* | the singleton set, consisting only of the number 7 |
| 15..19 | *is* | the set of all numbers between 15 and 19 including end points |
| 1..3, 7, 15..19 | *is* | the set theoretic union of the three previous sets i.e., the seperator "," acts as set theoretic union |

If the $C1$-end of an association $ass$ between $C1$ and $C2$ carries as multiplicity the subset $M$ of natural numbers any snapshot has to satisfy the consistency requirement that for every element $c_2$ in $C2$ the number of element from $C1$ in relation to $c_2$ occurs in $M$. That is: look at the set $\{c_1 \in C1 \mid ass(c_1, c_2)\}$. determine its cardinality $n$ and check wether $n$ is in $M$. If the multiplicity is 1, thus $M = \{1\}$, then there has to be for every $c_2 \in C2$ exactly one element $c_1 \in C1$ satisfying $ass(c_1, c_2)$.

The same applies for multiplicities attached to the $C2$-end.

### 2.2.3 Comments

Role names get lost in this representation.

Names for associations are optional. In case there is no name we use role-name1 - rolename2 instead. If the name *review* had been omitted in Figure 2.2 we would have called it the *referee-paper* relation.

Sofar we have only considered binary association, i.e. associations with just two associations ends. A association with n association ends is interpreted as an n-ary relation in any snapshot. All that was said above for binary relations carries over to n-ary relations.

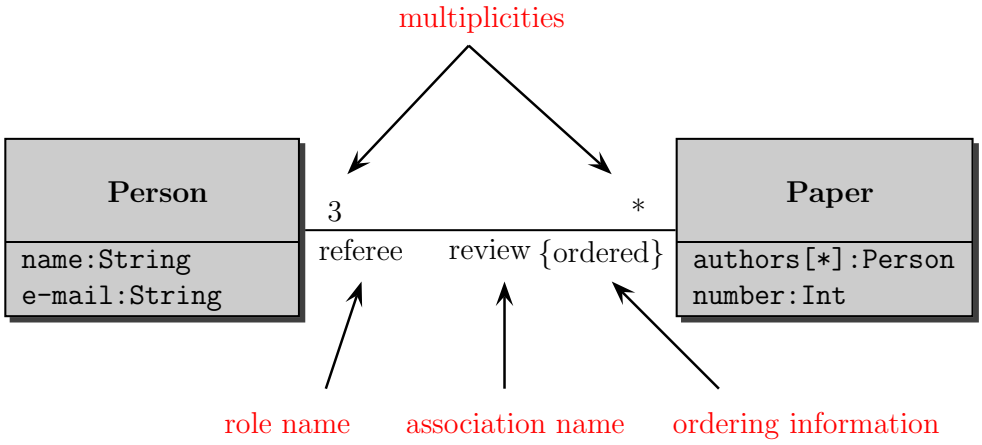



Figure 2.3: Review association with ordering

## 2.3 Role names

### 2.3.1 Example

### 2.3.2 Semantics

As pointed out in the comments in Section 2.2 role names get lost, when interpreting an association as a relation. For this reason we present here another way to look at associations and their association ends.

A binary association $ass$ between classes $C1$ and $C2$ gives rise to two functions $f1$ and $f2$. The first with domain $C1$ and the other with domain $C2$. The range of function $f1$ depends on the multiplicity and further adornment attached to the association end at $C2$. If the multiplicity is 1 then $f1 : C1 \to C2$. If the multiplicity is $*$ then $f1 : C1 \to Set(S2)$. If the multiplicity is $n$ for some $n \in \mathbb{N}$ then $f1 : C1 \to Set_n(S2)$. Other multiplicities are handled correspondingly. The same applies, of course, to $f2$. When we comsider a particular class diagram the abstract notion, $f1$ and $f2$, will be replaced by the role names attached at the $C2$ respectively at the $C1$ end of the association.

The *review* association in Figure 2.2 thus gives rise to the two functions

$$\begin{array}{lcll} referee & : & Paper & \rightarrow & Person \\ paper & : & Person & \rightarrow & Set(Paper) \end{array}$$

The UML Standard offers an ordering attribute for association ends with the possible values *unordered*, this is the default, *ordered* and *sorted*. Figure 2.3 shows an example. In this case the range of the associated function is not the set of subsets of the target class, but the set of (finite) sequences. In Figure 2.3 the function *paper* will be of signature $paper : C1 \rightarrow Seq(C2)$.

An $n$-ary association will give rise to $n$ functions in the same way as has been described for binary relations.

If an association end carries an *ordered* label the UML Standard says nothing about how to communicate what ordering should be used. This information has to be obtained somehow. In Figure 2.3 it is plausible to order the papers according to their *numer* attribute.

### 2.3.3 Comments

Since role names are optional we also have to make provisions in case they are missing. In this case the name of the class attached at the particular association end is used as a role name, in lower case letters.

Interpreting an association in the way described in this section the name of the association does not appear anymore. The fact that the functions $f1$, $f2$ arise from one and the same association has to be expressed as a constraint on snapshots of the diagram. For the functions $referee$ and *paper* in the diagram of Figure 2.2 these constraints read:

For every Person $r$ and every paper $p$, if $p$ is an element of $paper(r)$ then $r$ is an element of $referee(p)$ and vice versa: i.e. if $r$ is an element of $referee(p)$ then $p$ is an element of $paper(r)$.

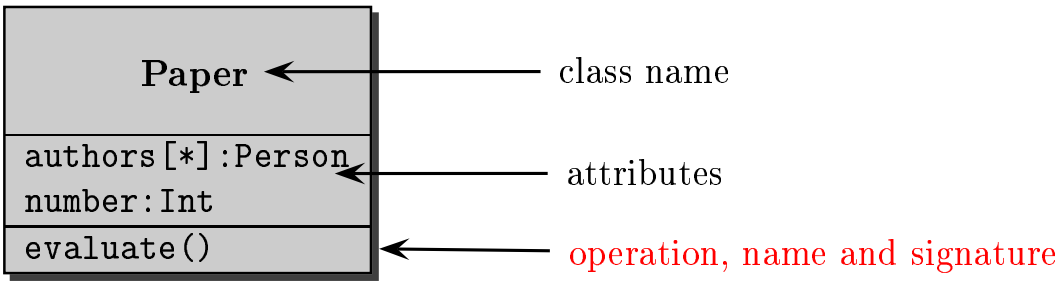



Figure 2.4: Class with operation

## 2.4 Operations

### 2.4.1 Example

### 2.4.2 Semantics

According to the UML semantics description operations are *conceptual constructs*. Operations are services that can be requested from an object of their class and will eventually be implemented. An implementation of an operation is called a method. Operations in a class diagram fix the number and types of the actual parameters of their implementation and also the type of the return value if it exists.

In our semantics operations are interpreted as transitions from one snapshot to one or in the case of non-deterministic operations to more than one successor snapshot. Set theoretically an operation is thus interpreted as a set of pairs of snapshots.

A particular case of operations are queries, i.e. operations that do not have side effects. They are treated differently in UML/OCL in that they may occur within OCL expressions. We interprete queries as partial functions in all snapshots having in addition to the specified arguments one more argument, whose type is the class of the operation. The only consistency requirements are that the declared argument and value types of an operation and its implicit argument are respected. Note, the meaning of operations with side effects cannot be defined by a function.

### 2.4.3 Comments

## 2.5 Subclasses

### 2.5.1 Example

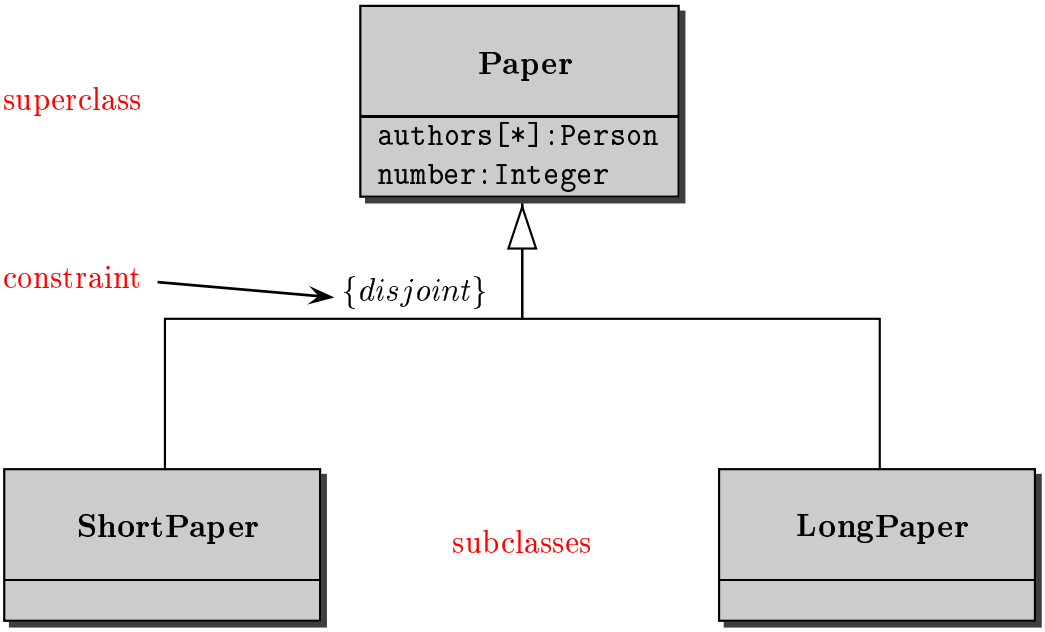



Figure 2.5: Subclasses

### 2.5.2 Semantics

If $C1$ is a subclass of $D$ in a class diagramm $CD$ then in any snapshot of $CD$ $C1$ is a subset of $D$. In general there is no further requirement, i.e. if $C1$ and $C2$ are all subclasses of $D$ then $C1$ and $C2$ need not be disjoint nor is the union of $C1$ and $C2$ required to be all of $D$. UML allows to add the following constraints to the subclass relationship (see [OMG, 2000b, page 2-36])

- complete
- disjoint
- incomplete
- overlapping

If a subclass relationship carries the constraint *disjoint* then we require that in any snapshot its subclasses will be interpreted as mututally disjoint sets. If it carries the constraint *complete* then we require that the union of its subclasses equals the superclass.

In any snapshot of the class diagram shown in Figure 2.5 there is no common element of the sets ShortPaper and LongPaper and there may be elements in Paper that do not belong neither to LongPaper nor to ShortPaper.

### 2.5.3 Comments

In [Evans *et al.*, 1999] the subclass relationship is also translated into the subset relation.

The *completeness* requirement:

> If $C1, \ldots, Ck$ are all subclasses of $D$ then we also require in any snapshot that $D$ is the union of $C1$ and $C2$ and $\ldots Ck$.

It looks innocuous but has dramatic consequences. If $C$ is the only subtype of $D$ in a class diagram then $C$ and $D$ would have exactly the same elements in all snapshots. As a further consequence multiple inheritance would also be problematic. If $C$ is the only subclass of both $D1$ and $D2$ then in all snapshots $C$, $D1$, and $D2$ would coincide.

If $C$ is a subclass of $D$ then in any snapshot all attributes of $D$, being interpreted as functions on $D$, are also defined on $C$ and yield of course the same value. Thus the above semantics implies inheritance of attributes in a very strong sense. This seems however reasonable.

The concept of subclass is understood as direct, or one-step subclass. If we want to talk about subclasses of subclasses, and so on, we use the term hereditary subclass.

There are people supporting a dissenting vote, that subclasses should not be repesented as subsets, but rather be interpreted in a broader sense that a subclass *behaves like* its superclasses. See e.g. [Bittner & Koch, 2000].

## 2.6 Abstract Classes

### 2.6.1 Example

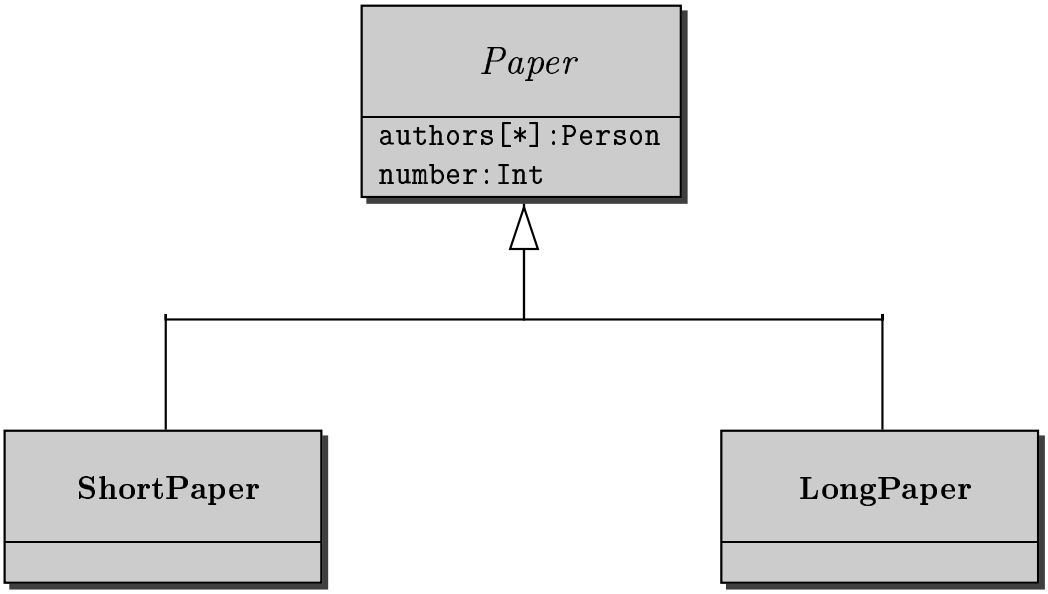



Figure 2.6: An abstract class with subclasses

The class name of an abstract class is typeset in *italics*.

### 2.6.2 Semantics

An abstract class is interpreted as the disjoint union of all its subclasses. If a subclass is again abstract this explanation is to be understood recursively. An abstract class without subclasses does not have any elements in all snapshots.

### 2.6.3 Comments

There is not much of a distinction between a class and an abstract class in our semantics. The usual explanation that an abstract class does not have direct elements but may have indirect elements, does not make sense in a set theoretic setting. Compare the quotation from [Rumbaugh *et al.*, 1998, page 114].

> The distinction between modeling a class as abstract or concrete is not as fundamental or clear-cut as it might first appear. It is more a design decision about a model than an inherent property.

The notion of a direct element does not exist in set theory. One could make it precise in the following way: Let a collection of sets be given and $S$ one of them. An element of $S$ is called a direct element of $S$ if there is no subset of $S$ containing it.

From the semantics point of view the only difference between the class diagrams in Figures 2.5 and 2.6 is the disjointness requirement in the former.

## 2.7 Class Attributes

### 2.7.1 Example

The class attributes, also called class scope attributes or static attributes, are shown by underlining their name and type, see [Rumbaugh *et al.*, 1998, page 169].

### 2.7.2 Semantics

The idea is that class attributes are not attached to the instances of a class but to the class as a whole. How does this fit into our semantic framework? There are three possibilites

1. We present the most systematic version first. In Chapter 4 we will come to know the meta-type (or as you might also say, meta-class)

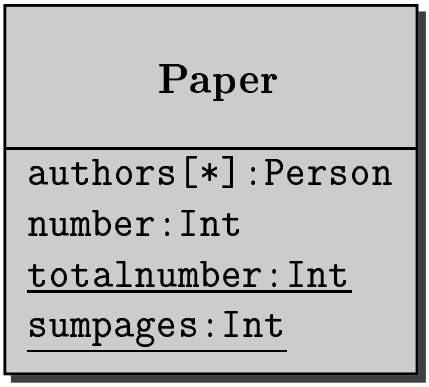



Figure 2.7: A class with class scope attribute

*OclType*, which will in any snapshot of a class diagram $CD$ contain, among others, all classes in $CD$ as elements. Class attributes will be interpreted as partial functions on *OclType*. We require that a class attribute attr in class $C$ will only be defined for $C$ and its subclasses and undefined for all other elements of *OclType*.

2. The second version proceeds as in the case of normal (i.e. instance scope) attributes, but adds as an additional constraint that the value of the function attr is the same for all elements in $C$.

3. We favour this third version. A class attribute attr:T of a class $C$ is interpreted as an element in $T$. Reference to this element is made through a constant *C.attr*. Note, that *C.attr* is viewed as one token, not as a function application.

### 2.7.3 Comments

It is apparent from the previous section that class attributes do not fit well into an object-oriented world. They are, by definition, not attached to an object. Either one involves the meta-level, where classes themselves now are objects or one departs in some way from the object oriented view.

Figure 2.7 shows two typical class attributes. *Paper.totalnumber* is supposed to hold the total number of papers received at any given time during the

submission process, while *Paper.sumpages* gives the sum of the number of pages.

## 2.8 Association Class

### 2.8.1 Example

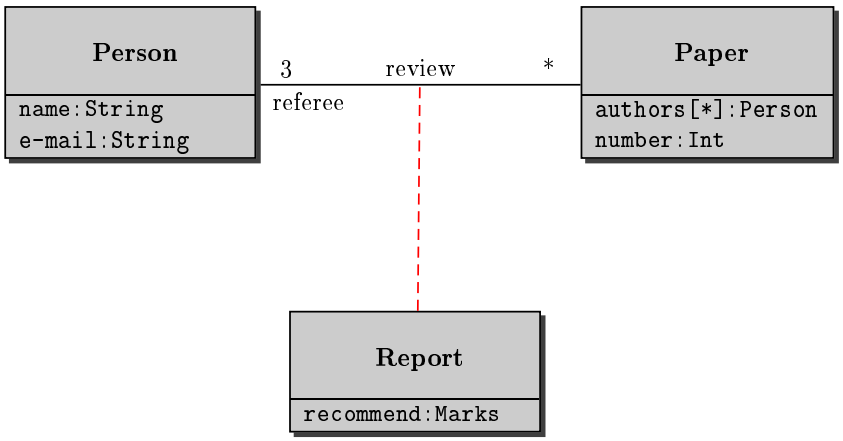



Figure 2.8: An association class

### 2.8.2 Semantics

An association class is conceived as both a class and an association. In any snapshot an association class attached to a binary association is interpreted as a set of pairs. An association class attached to an n-ary association is interpreted as a set of n-tupels. The consistency requirements of the association an association class is attached to carry over to the class itself.

### 2.8.3 Comments

One would expect that there should be some way to access the first and second entry of an instance of an association class. And indeed there is, as will be explained in Chapter 4 on OCL.

The main reason for introducing association class is that one may attach attributes to instances of an association. The attribute *recommend* in Figure 2.8 is, as explained in the section on attributes, in any snapshot interpreted as a function of its class, in this case *Report*, to its target class, in this case the enumeration class $Marks$. Strictly speaking *recommend* takes just one argument, but since this argument is a pair of elements, we may think of *recommend* as the UML-way to speak about a binary function.

## 2.9 Data Types

### 2.9.1 Example

### 2.9.2 Semantics

The interpretation of a data type is the same in *all* snapshots. Data types have no attributes. All operations defined on data types are *queries*, i.e. they do not have any side effect.

### 2.9.3 Comments

The intention of the meaning of data types is explained in [Rumbaugh *et al.*, 1998, page 247]

> A data type is a descriptor of a set of values that lack identity (independent existence and the possibility of side effects).
> ⋮
> Their semantics are mathematically defined outside the type-building mechanisms in a language.

In our set theoretic setting a *set of values lacking identity* is made precise as a set whose interpretation is the same in all snapshots and has no attributes.

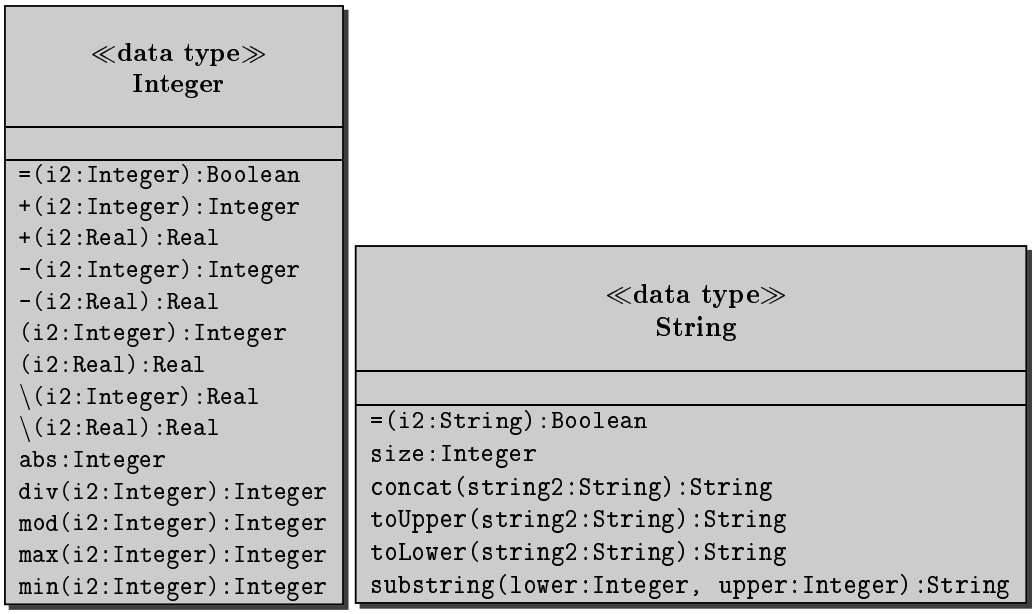



Figure 2.9: Data types

It is not possible to create new instances of a data type. The indentity of an data value is just its value, no further distinctions by attributes are possible.

The standard document describes in [OMG, 2000b, Section 2.7] only the data types used for defining the metamodel of UML.

At the time of writing this text it is not clear whether the data types used in the UML metamodel are meant to coincide with the data types in OCL. In Figure 2.10 the OCL data types for integers and strings are shown. The interpretation of the data types *Integer*,*String*, *Real* is understood in the mathematical sense. In particular they are infinite sets. Further refinements of data types to *language types*, i.e. data types defined in the syntax of a particular programming language are possible, see [Rumbaugh *et al.*, 1998, page 323].

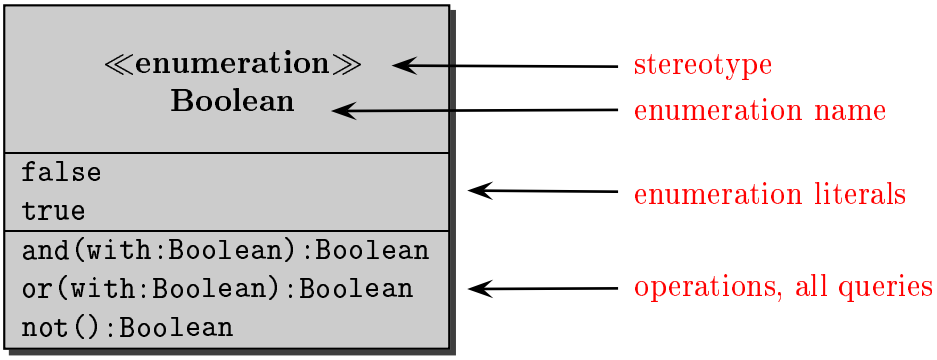



Figure 2.10: An enumeration type *Boolean*

## 2.10 Enumerations

### 2.10.1 Example

Enumerations are special user-defined data types. They are distinguished from other classes by the string ≪ *enumeration* ≫ in the name compartment, see e.g. Figure 2.11. Additional tags like ≪ *enumeration* ≫ are called *stereotypes* in UML. Stereotypes are used to extend or modify the semantics of UML model elements but should not change the structure of pre-existing model elements.

### 2.10.2 Semantics

In any snapshot an enumeration class is interpreted as the set containing exactly the enumeration literals listed in its attribute compartment. Notice, the interpretation of an enumeration class, like all other data types does not change. It remains constant in *all* snapshots.

### 2.10.3 Comments

Another example of an enumeration type, taken from [Rumbaugh *et al.*, 1998, page 268], is shown in Figure 2.10. Note, that enumeration types may also

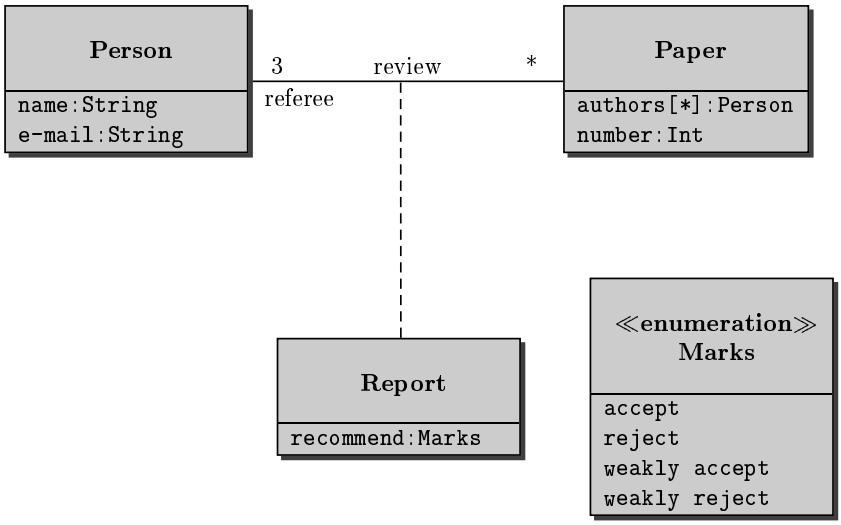



Figure 2.11: An enumeration class

contain operations. It is required though, that all operations be queries, i.e. have no side effects.

## 2.11 Aggregations and Compositions

We return to section 2.2 and take a closer look at the concept of associations. The ends of an association may be adorned with one of the labels

- none
- aggregate
- composite

The well-formedness of UML diagrams requires that at most one end of an association carries a label different from *none*.

### 2.11.1 Example

*none* is the default and usually omitted. An aggregation is shown as a hollow diamond adornment at the corresponding end of the association. A composition is shown as a solid-filled diamond adornment.

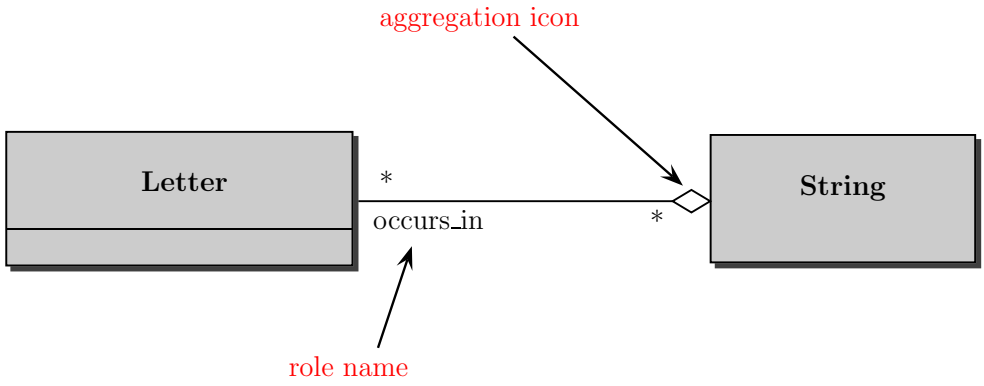



Figure 2.12: An aggregation association

### 2.11.2 Semantics

On the level of our mathematical semantics there is no differecne between an arbitrary associations and an aggregation or composition. In any case the association is a set of pairs.

There are intentional differences though. Aggregation and composition are used to represent the relationship between a whole and its parts. The parts of an aggregation may be used multiply. In contrast, if an object is used in a composition it is *consumed*; it cannot appear twice in the same or another composition. A typical example of an aggregation are train itineraries. A city, or station, may occur in many itineraries without being consumed. An example of a composition are necklaces made of beads: here an individual bead can occur only once and only in one necklace. Another example is shown in Figure 2.13

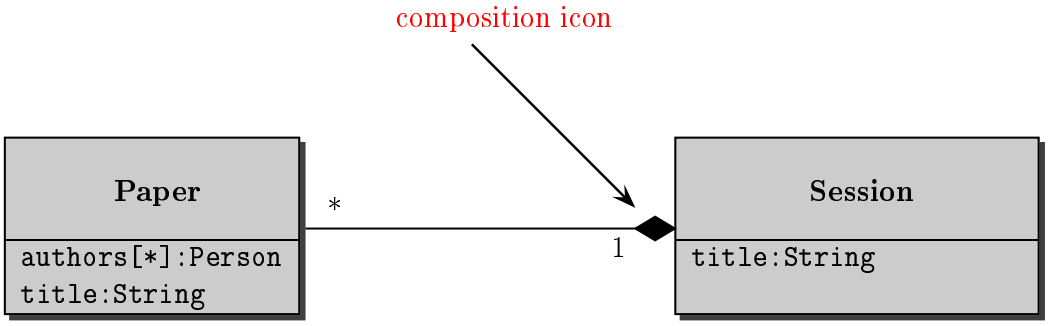



Figure 2.13: A composition association

As remarked there is no difference in the mathematical representation of an arbitrary association and an aggregation or composition on the level of an individual association. On the level of UML diagrams that may contain many aggregations and compositions one usually requires that there is no cycle of associations of these kinds. This is to avoid the situation that an object is part of itself. Also consecutive aggregation and composition links are required to be transitive: if object $a$ is part of a part of $b$, then $a$ is also a part of $b$.

### 2.11.3 Comments

We quote from [Rumbaugh *et al.*, 1998, page 148]

> In spite of the few semantics attached to aggregation, evrybody thinks it is necessary (for different reasons). Think of it as a modeling placebo.

A very typical situation involving aggregation and subclasses is the class diagram for the composite pattern shown in Figure 2.14. It shows two kinds of components, composite components and those that cannot be be further decomposed, here called leaves. If a composite object $c$ is an aggregation of the components $c_1, \ldots c_k$, the $c_i$ are called the children of $c$. This terminology

suggests to image composite objects as trees. The children of a composite object may themselfs again be composite or they may be leaves. For a full description of the composite pattern, see [Gamma *et al.*, 1995, page 164]

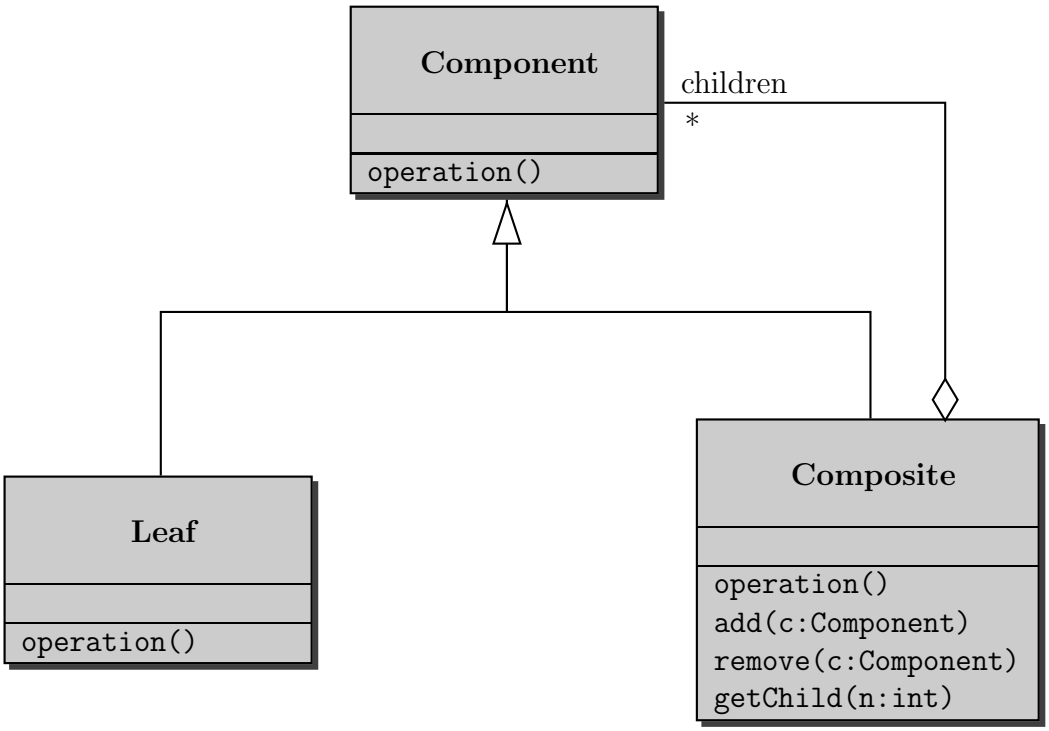



Figure 2.14: A composite pattern

## 2.12 Qualifiers

### 2.12.1 Example

Qualifiers are drawn as rectangles below or to the side of a class box. They should be smaller in size then the attached class. Within the qualifier box one or more attributes may occur. The syntax qualifier attributes is the same than that of class attribute. The only difference is that qualifier attributes cannot have initial values. The qualifier box and the qualifier attributes are part of the associations, not of the class next to them. Qualifiers may only occur in binary associations. They may occur at both association ends simultaneously, though this is an exceptional case.

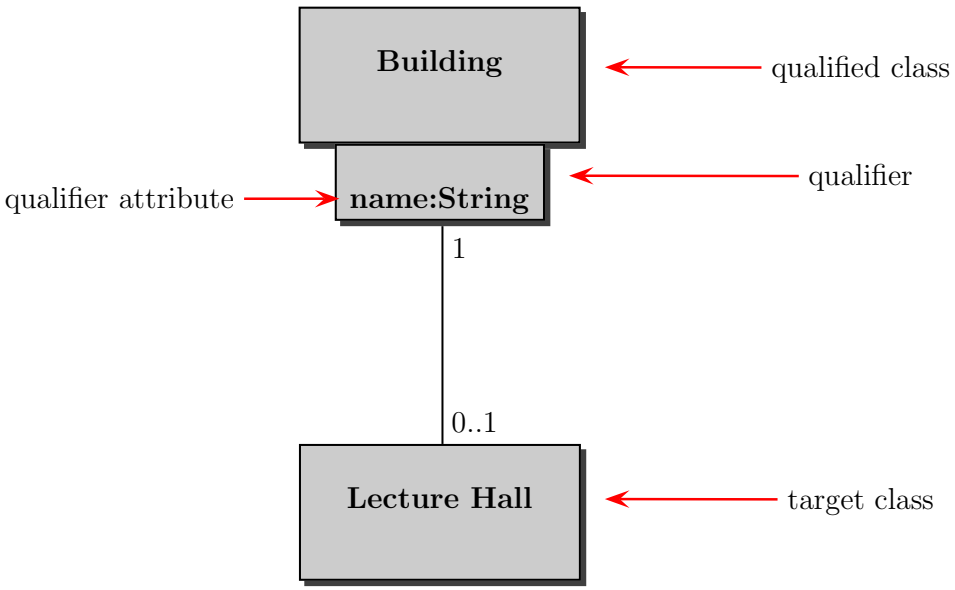



Figure 2.15: Association with qualifier

### 2.12.2 Semantics

The association in Figure 2.15 associates lecture halls with buildings. Every lecture hall is, of course, housed in a unique building. One the other hand, there may be more than one lecture hall in one building. If we name a building and a name then there is no lecture hall in this building by that name or there is exactly one. This is part of the information contained in diagram 2.15.

The qualified association in Figure 2.15 may e.g. be represented as two functions. One function, describing the part from bottom to top, let us call it *location* associates with every lecture hall a building an a name, $location$ : Lecture Hall $\rightarrow$ Building $\times$ *String*. Another function, let us call it *hall* is a two place function associating with every pair of building and string a lecture hall. $hall$ : Building $\times$ *String* $\rightarrow$ Lecture Hall This is a partial function.

### 2.12.3 Comments

After all, we see that there are ways to express functions with more than one argument in UML.

# Chapter 3

# UML Object diagrams

Object diagrams are another means, in addition to class diagrams, for modeling the static view of a system. It is very important to understand that they operate on a level different from the level of class diagrams. The nodes in object diagrams are individual object. In the object diagram shown in Figure 3.1 five object exists, three of class *Person* and two of class *Paper*.

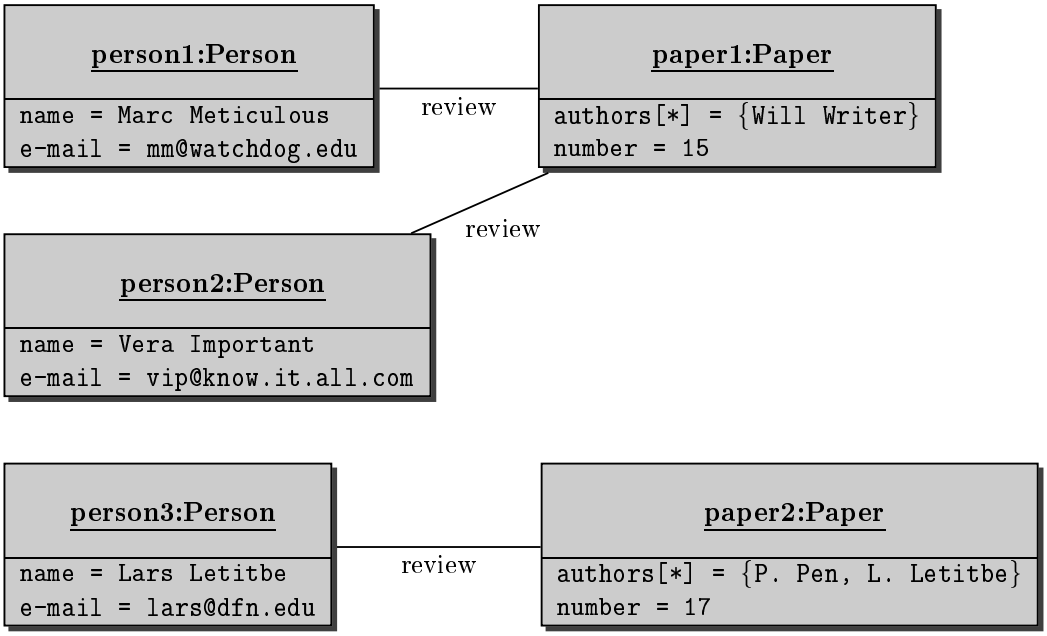



Figure 3.1: An object diagram

Attributes in object diagrams are assigned definite values and also the association among objects are shown.

While class diagrams limit potential snapshots of the modeled system, object diagrams represent snapshots. We will infact from now one make no difference between snapshots and object diagrams and will use these terms interchangably.

At the level of object diagrams multiplicities at associations ends are always 1.

It makes sense to ask wether an object diagram $D_{obj}$ conforms to a given class diagram $C$. That is to say

- Are the values of attributes in $D_{obj}$ of the type specified in $C$?
- Does $D_{obj}$ observe the multiplicity constraints of $C$?
- Are the associations occuring in $D_{obj}$ declared in $C$?

In this sense the object diagram in Figure 3.1 conforms to the class diagram in Figure 2.2. We also notice that Lars Letitbe reviews a paper co-authered by himself. Certainly an indication that further constraints should be added.

Object diagrams are used much less than class diagrams.

In Figure 3.2 we see two further examples of object diagrams. The lefthand diagram conforms to the class diagram 2.14, while the righthand diagram does not. It contains a *child*-association between leafs, which is not specified in Figure 2.14. The object diagram on the lefthand side may be viewed as the three-element list $\langle element1, element2, element3 \rangle$. The diagram in Figure 2.14 requires that the child object element2 associated with element1 is of type Component. element2 is declared to be of type Composite. Since Composite is a subclass of Component this constraint is satisfied.

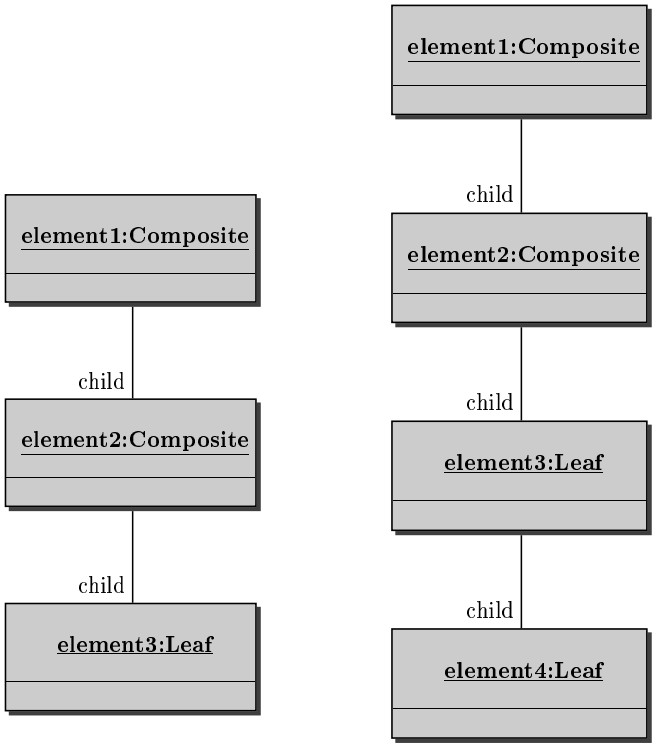



Figure 3.2: An object diagram representing a list

# Chapter 4

# OCL by Example

The Object Contraint Language (OCL) is part of the UML Standard, [OMG, 2001, Chapter 6]. An easy introduction is available through the book [Warmer & Kleppe, 1999]. Material on a precise semantics of OCL is contained in the volume [Clark & Warmer, 2002]. See in particular the contribution [Gogolla & Richters, 2002].

OCL was introduced to express subtleties and nuances of meaning that diagrams cannot convey by themselves. It was first developed in 1995 by Jos Warmer and Steve Cook. The most extensive use of OCL so far is within the UML standard itself, where it is used in the semantics description of the UML meta model.

OCL is percieved by its creators as a *formal* language. On the other hand they put emphasis on the fact that OCL is not designed for people who have a strong mathematical background. We quote from [Warmer & Kleppe, 1999, Preface]

> The users of OCL are the same people as the users of UML: software developers with an interest in object technology. OCL is designed for usability, although it is underpinned by mathematical set theory and logic.

We will first present some of the basic features of OCL expressions by example. Then, in a second step, attempt a systematic description.

## 4.1 Contexts

Every OCL constraint needs a UML diagram $\mathcal{D}$ it refers to. Without $\mathcal{D}$, constraints cannot be formulated let alone their meaning be determined. We refer to $\mathcal{D}$ as the context diagram of an OCL constraint. OCL constraints can be attached to every model element in $\mathcal{D}$. We will only treat the case where constraints are added to a class diagram. This is by far the most frequently occuring case. Two basic context modes need to be distinguished

- The classifier context.
- The operator context.

The general form of a constraint in classifier context is

**context** ( $c$ **:**)? typeName
  **inv** expressionName? **:** OclExpression

The trailing question mark ? indicates optional elements; OCL keywords are set in **boldface**. 'typeName' will typically be the name of a class in the fixed UML diagram. We will explain other possibilities later. It is possible to introduce a name for easy referencing of expressions. The optional parameter $c$ will act very much like a variable of the type given by type name in the following OCL expression. Variable is here to be understood in the way it is used in formal logic. A header may define more than one expression:

**context** ( $c$ **:**)? typeName
  **inv** expressionName$_1$? **:** OclExpression$_1$
  ...
  ...
  **inv** expressionName$_n$? **:** OclExpression$_n$

Constraints for an operator context look like this:

**context** ( $c$ **:**)? typeName **::**opName(**(**$p_1$**:** type$_1$**;** ... **;**$p_k$**:** type$_k$ **)**:rtype
  {**pre**,**post**} expressionName? **:** OclExpression

Here opName is meant to be the name of an operator defined on the given class. The list of parameters $p_1 \ldots p_k$ may be empty and the return type, rtype, may be missing or both. As above, an operator constraint may contain more than one expression.

### 4.1.1 Comments

In the headers just shown OCL expressions have to be of type Boolean. Also the stereotype **inv** can only appear in a classifier context while the stereotypes **pre** and **post** can only show up in operator contexts.

We have added the optional parameter( $c$ **:**)? also in the operator context to set it on equal footing with classifier contexts, though we have seen no example of this in the literature.

## 4.2 Constraints with Attributes

### 4.2.1 Example

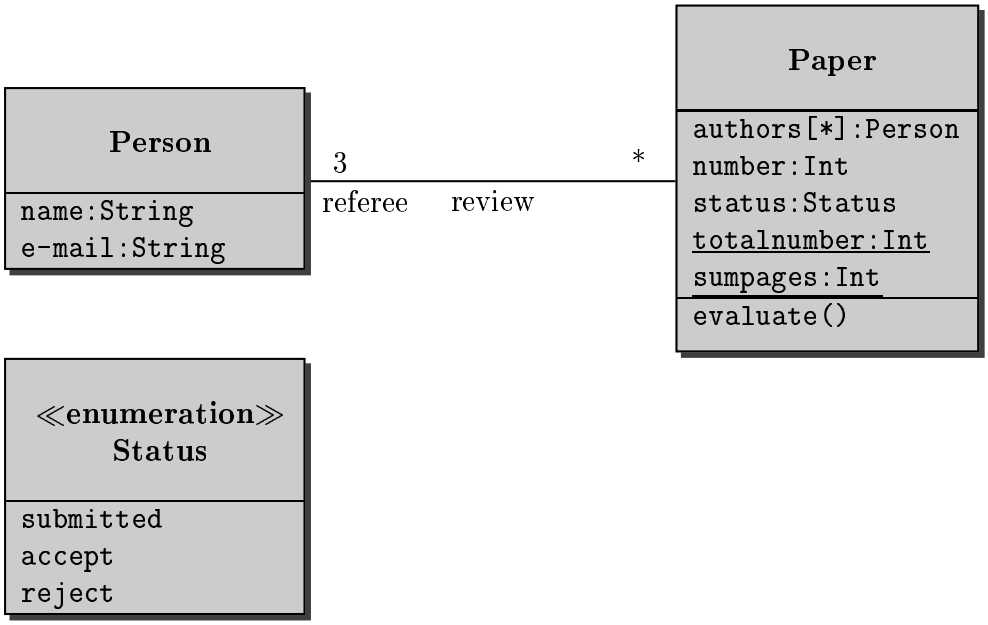



Figure 4.1: Context diagram for attribute constraints

Here is our first and simple example of an OCL contraint in the context of the diagram in Figure 4.1. Since a diagram usually contains more than one model element a further focus is needed. This is provided by the *header*.

**context** Paper
  **inv** number $\geq 1$

In our example the header contains a class name, here *Paper*. The attributes used in the constraint then have to be attributes declared in this context class. The actual constraint is prefixed by the key word **inv**. This signifies that the constraint is to be an invariant of its context class.

Equivalent notational variations of the constraint in classifier context are:

**context** Paper
  **inv self**.number $\geq 1$

**context** c:Paper
  **inv** c.number $\geq 1$

**context** c:Paper
  **inv** startCount : c.number $\geq 1$

**context** Paper
  **inv** startCount : number $\geq 1$

Here is an example of an operator constraint for the same context diagram (see Figure 4.1).

**context** *c*:Paper::evaluate()
  **pre** c.status = submitted
  **post** c.status = accept or c.status = reject

This constraint states pre- and postconditions for the operation *evaluate* in class *Paper*.

Variations, as shown above, are also possible for operator constraint.

### 4.2.2 Constraint Syntax

The syntax mimics access to attributes in object-oriented programing languages: a designator for an object followed by a dot followed by the name of the attribute. The default designator is **self**.

### 4.2.3 Meaning of the Constraint

The meaning of the example constraint in classifier context is that the number attribute of any object should be greater or equal to 1. In colloquial terms: the numbering of the papers in our scenario starts with 1.

More precisely an attribute constraint of the general form $relation(c.attr)$ is evaluated in an object diagram $D_{obj}$. It is true, if for all object $o$ of the context class the value $o.attr$ of the attribute $attr$ in $D_{obj}$ satisfies the constraints, set up by $relation$. Thus **self** or $c$ may be viewed as (implicitely) universally quantified variables.

The meaning of the above example of a constraint in operator context says that in the state reached after calling the $evaluate()$ method the value of the attribute $status$ will be one of $accept$ or $reject$. But, this can only be guaranteed when the precondition is satisfied. In this example the preconditions asks that before calling $evaluate$ the value of the $status$ attribute equals $submitted$. If $c.status$ is $undef$ then no claim is made what will be true after calling $evaluate$. Even trickier, the specification as it stands, does also make no claim what happens, when $evaluate$ is called from an object $c$ in a state where $c.status$ equals $accept$.

Pre- and postconditions are seen as part of a contract. The method agrees that after its execution the postcondition is true. But, it is the obligation of the caller of a method to ensure that the precondition holds. The OCL standard does not make any commitment what should happen, when the contract is brocken. A thorough discussion of the possible variation points in the semantics of pre- and postconditionsis contained in [Hennicker *et al.*, 2001] .

This explanation of the meaning of pre- and postconditions avoided the question, what is the meaning of an operation itself. We will come back to this question in Section 4.11.

### 4.2.4 Comments

The UML standard does not mention the problem of termination. Is it assumed in general that every method call always terminates? Or, if we state a postcondition, does that involve the claim that the method terminates, or at least terminates in all states where the precondition is satisfied? We take the position that no commitment to termination is involved with the pre- and postconditin concept. This is what is called the *partial correctness* notion in the literature. The situation is different, if an operation *op* is declared a *query*, by using the stereotype $\ll$ *query* $\gg$. Then it is assumed that *op* terminates on every input.

## 4.3 Types

OCL is a typed language. Every expression has a uniquely determined type. The syntex of expressions is restricted by typing rules. We distinguish the following types

1. Model types
   Every class form the context diagram of an OCL constraint is a type.

2. Basic types
   There are 4 basic OCL types: *Integer*, *Real*, *Boolean* and *String*

3. Enumeration types
   These are user defined types.

4. Collection types
   The collection types in OCL are *Set*, *Bag*, *Sequence*.

5. Special types
   Special types are a tricky issue in OCL. We mention here only the type *OclAny*. A complete listing can be found in Section 12.4.

Subtyping is also part of the OCL type concept. For type expressions $T_1, T_2$ the direct subtype relation $T_1 < T_2$ is defined by the following rules:

1. If $T_1, T_2$ are model types then $T_1 < T_2$ holds exactly when in the contex UML diagram $T_1$ is a subclass of $T_2$.

2. $Integer < Real$.

3. For all type expressions $T$, not denoting a collection type,

   (a) $Set(T) < Collection(T)$

   (b) $Bag(T) < Collection(T)$

   (c) $Sequence(T) < Collection(T)$

4. If $T$ is a model, basic or enumeration type then
   $T < OCLAny$.

5. If $T_1 < T_2$ and $C$ is any of the type constructors $Collection$, $Set$, $Bag$, $Sequence$, then $C(T_1) < C(T_2)$.

The subtype relation denoted by $\ll$ is the transitive, reflexive closure of the direct subtype relation $<$ If $T_1 \ll T_2$ holds, we also say that $T_1$ conforms to $T_2$.

### 4.3.1 Example

Looking at Figure 4.1 we discover the following types:

- We fix the context **context** p:Person

  Then the expressions $p.name$ and $p$.e-mail have as type the basic type $String$.

- In the context **context** c:Paper

  Then $c.number$ has basic type $Integer$, $c.status$ is of model type $Status$, $c.authors$ is of collection type $Set(Person)$.

### 4.3.2 Syntax

Type expressions, that is expression that denotes types, are rather simple in OCL. Basic, model, enumeration and special types are denoted by their names. If $T$ is a type expression that this not a collection type then $Collection(T)$, $Set(T)$, $Bag(T)$, $Sequence(T)$ are also type expressions.

Note, that $Set(Set(T))$ is not a legal type expression.

The UML standard gives a complete listing of the operations available for the built-in types of OCL, i.e. the basic, special, and collection types. This listing also fixes the typing information of these operations. The typing information for attributes and associations (see the next section) are given in the constex UML diagram. For this reason types appear not very often explicitely in OCL expression. They do in variable declarations.

### 4.3.3 Meaning of Types

In a snapshot types are interpreted as set of objects plus an additional symbol $\perp$, that denotes $undefined$.

If $T$ is a class in a context diagram $CD$ and consequently a model type in OCL, then in any snapshot of $CD$ the meaning of $T$ is the set of all objects in the class $T$ in this snapshot. The meaning of the basic types is as one would expect. Note, the interpretation of these types is independent of any particular context diagram or snapshot. The meaning of an enumeration type $T$ is also evident, it is the set of its literals.

If a type $T$ is interpreted in a snapshot by a set $M$ of objects then $Set(T)$, $Bag(T)$, $Sequence(T)$ are interpreted as the set of subsets of $M$, of all bags of elements from $M$, of all sequences of elements from $M$. The interpretation of $Collection(T)$ is the union of o the interpretations of $Set(T)$, $Bag(T)$, $Sequence(T)$.

That leaves for the moment only the special type $OCLAny$ to be explained. The interpretation of $OCLAny$ in a snapshot is the set of all objects contained in one of the sets interpreting a model, basic or enumeration type.

### 4.3.4 Comments

It has been one of the basic design decisions of OCL to forbid nested set operations. The main motivation was to keep things simple. There are proposals to drop this restriction.

Notice, that $OclAny$ despite its name is not a supertype of everything. Collection types are not subtypes of $OclAny$.

## 4.4 Constraints with Associations

### 4.4.1 Example

This constraint refers to the context diagram in Figure 4.2.

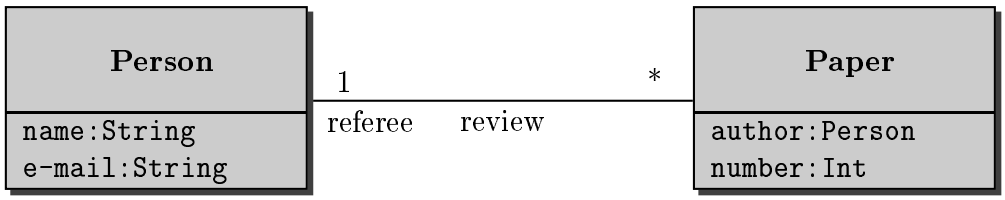



Figure 4.2: Simplified context diagram for association constraints

**context** c:Paper
  **inv** c.author <> c.referee

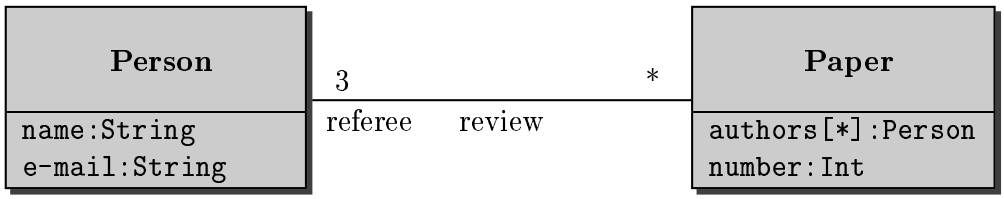



Figure 4.3: Context diagram for association constraints

The following constraint refers to the context diagram in Figure 4.3.

**context** c:Paper
  **inv** c.author -> intersection(c.referee) -> isEmpty

### 4.4.2 Constraint Syntax

The variable **self** or *c* may be followed by a dot and the name of an association. Depending on the multiplicity of the association the evaluation will be a single object or a set of object. There are a number of built-in operations

on sets in OCL, *intersection* and isEmpty are among them, see 12.3 for a complete listing. If *exp* is an OCL expression that evaluates to a set $M$, the application of an operation with name *op* to $M$ is then denoted by *exp* -> *op*.

### 4.4.3 Meaning of the Constraint

Consider an expression of the form $c.assoc$ (or **self**.$assoc$). To determine a meaning we need an object diagram $D_{obj}$ and an assigment of an object $a$ to $c$. Then the meaning of $c.assoc$ is the set of all objects $b$ such that $a$ and $b$ are related by $assoc$ in $D_{obj}$. OCL is not very systematic here: if the multiplicity of $assoc$ is 1, the the result of the evaluation is not a one-element set, but the element itself.

If the source-end of the association $assoc$ carries the stereotype $\ll ordered \gg$ the meaning of $c.assoc$ is the sequence of all objects $b$ such that $a$ and $b$ are related by $assoc$.

### 4.4.4 Comment

The use of the separator -> instead of the dot ".", is a purely syntactical device, meant to make it easy for the reader to spot the occurences of set operations. Assume that in a specification the -> symbols get inadvertently changed into dots. Then you would be able to restore the original text unambiguously. There is one tiny exception to this. Consider an OCL expression $self.f$ with $f$ an association with multiplicity [0..1] at the value-end of $f$. The standard [OMG, 1999a, Subsection 7.5.5] allows to access $self.f$ as a set or as an instance, i.e. $self.f.g$ and $self.f \rightarrow g$ are both well-formed expressions, where $g$ is a set operation (think e.g. of *isEmpty*). See, also [Fowler & Scott, 1997, Section 3.6.1]. We will not lose much sleep over this issue.

## 4.5 Navigation

### 4.5.1 Example

The following constraints refer to the context diagram in Figure 4.4.

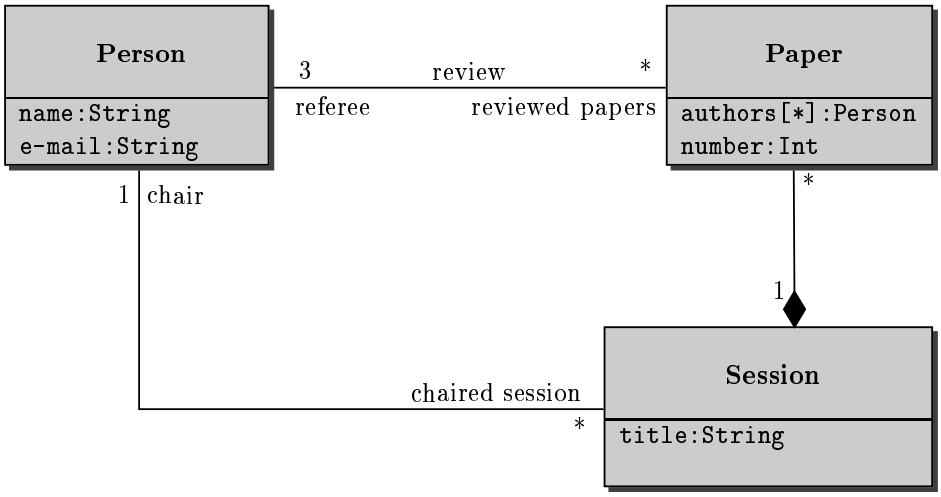



Figure 4.4: Constraints with navigation

**context** c:Paper
**inv** not(c.authors -> includes(c.session.chair))

**context** p:Person
**inv** p.reviewed_papers.session.chair ->includes(p)

The first constraint says, that no author or co-author of a paper presented at a session should be chair of this session.

The second constraint says, that every referee should be chair of a session that contains a paper he reviewed.

### 4.5.2 Constraint Syntax

If $e$ is an OCL-expression, that evaluates in a single object or a set of objects of class $C$, and $assoc$ is an association between class $C$ and another class, say $C_1$, then $e.assoc$ is a legal OCL expression. This process can be iterated.

### 4.5.3 Meaning of the Constraint

Consider an expression of the form $c.a_1.a_2$. Let $D_{obj}$ be an object diagram and $o$ an object of the same type as $c$.

The simplest case occurs, when the target multiplicity of $a_1$ is 1. Let $o_1$ be the unique object associated with $o$ via $a_1$. Then the whole expression consists of objects that are related by $a_2$ to $o_1$.

Otherwise, let $M_1$ be the set of object related to $o$ by the association $a_1$. Then the meaning of the whole expression $c.a_1.a_2$ is the set of all elements that are related via $a_2$ to at least one elment in $M_1$.

The process of evaluating expressions following along repeated association links through a diagram has been called navigation. So far, we have learnt the basic principles of navigation. There is more to come.

We have already come across sets in OCL and in the last section also sequences did show up. There is a third kind of collections used in OCL: bags,

or multisets, as they are sometimes called. Sets, bags, sequence are refered to in OCL as collections and the three-element listing is exhaustive.

What does this have to do with navigation? Consider the second constraint from above

**context** p:Person
  **inv** p.reviewed_papers.session.chair ->includes(p)

and an object diagram $D_{obj}$ where $p$ is interpreted as $MrImportant$. $MrImportant$ was the referee, among others, of the papers *All about Nothing* and *More of the Same*. Both papers will be presented in the section *All or Nothing*. Evaluating the expression $p.reviewed\_papers.session$ by following the links in the associations $reviewed\_papers$ and $session$, we notice that session *All or Nothing* occurs twice.

For this reason the designers of OCL decided that the evaluation of an association applied to a set-valued or bag-valued expression will be a bag. What happens, when an association is applied to a sequence? Then the result will again be a sequence. What happens if the source is a bag and the target end of the association is label as ordered? Then the result should be a sequence.

### 4.5.4 Comment

Let us look again at the OCL expression $c.a_1.a_2$, already mentioned above. Assume $c.a_1$ evaluates to a set of objects and for every $d$ in this set, also $d.a_2$ evaluates to a set. Then one could imagine $c.a_1.a_2$ to evaluate to a set of sets. The designers of OCL decided against this. Sets of sets, where ever they arise, are immediatedly $f$lattened, e.g.. the set $\{\{1,3,5\},\{4,8,9\},\{2,6,10\}\}$ is turned into $\{1,3,5,4,8,9,2,6,10\}$. Of course, flattening a set $M$ results in a set very different from $M$, but usually it is the flattened set or bag one is interested in.

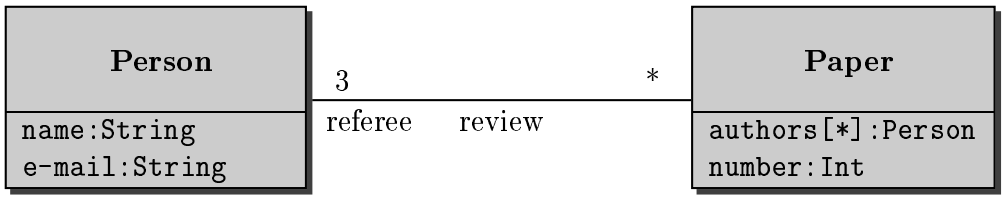



Figure 4.5: Context diagram for *allInstances*

## 4.6 allInstances

### 4.6.1 Example

In the context of the diagram in Figure 4.5 the following constraints are possible:

**context** Person
  **inv** Person.allInstances ->forAll(p | p.e-mail.size $\geq$ 3)

**context** Paper
  **inv** Paper.allInstances ->forAll(p1, p2 |
    p1 <> p2 implies p1.number <> p2.number)

The first constraint says that the e-mail address of every person is a string of length at least 3. The second constraints says that the numbering of papers is unique, i.e. different papers get different numbers. *forAll* is another set operator and corresponds to universal quantification, see 12.3.

### 4.6.2 Syntax

If $C$ is a type symbol then $C.allInstances$ is a legal OCL expression. In greater detail: *allInstances* is an operation on the predefined OCL type *OclType*. This is what is sometimes called a meta-type.

### 4.6.3 Meaning of allInstances

The interpretation of $OclType$ is the set of all types present in a fixed context. This includes model types as well as predefined OCL types. It is never mentioned in the standard and related documents, but $OclType$ should not be an element of it own interpretation.

Since $OclType$ is a concept on the meta-level, understanding its meaning requires familarity with abstraction. $OclType$ is **not** interpreted as the set of names of classes. Rather, every class is conceived as an abstract object. For the class $Person$ there is an abstract object $o_{Person}$, and this object is an element of the interpretation of $OclType$. $o_{Person}.name$ will evaluate to the string $Person$.

If $C$ a type symbol then the expression $C.allInstances$ evaluates in a snapshot $D$ to the set of all instances of class $C$ in $D$.

It is an (implicit) assumption of OCL that all occuring sets should be finite. For this reason it has been stipulated that expressions like $Integers.allInstances$ evaluate to $\perp$ (undefined). If $C$ is a collection type, say $C = Set(T)$ for a model type $T$, then $C.allInstances$ should also evaluate to $\perp$, since it would result in a set of sets of objects of type $T$, also not permitted in OCL.

The problems with $allInstances$ bear some resemblance to the problems with class attributes and operations addressed in Section 2.7. We prefer the following solution. For every model class $C$ a constant symbol (operation symbol with 0 arguments) $C.allInstances$ is available. As with class attributes we consider $C.allInstances$ as **one** token. This is in accordance with the position towards $allInstances$ in [Boldsoft *et al.*, 2002].

### 4.6.4 Comment

The concept of $OclType$ has always been controversial. The same applies to the operation $allInstances$ on $OclType$. Its use has been discouraged, see [OMG, 2001, Paragraph 6.5.11] and [Warmer & Kleppe, 1999]. There are OCL expressions where $allInstances$ is outright superfluous, as in the first example above, which can equivalently be written as

**context** p:Person
**inv** p.e-mail.size $\geq 3$

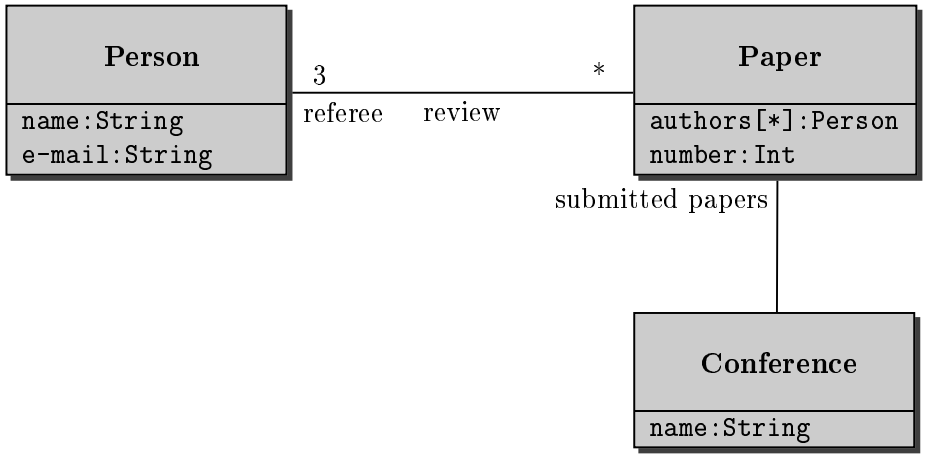



Figure 4.6: Expanded context diagram for *allInstances*

Avoiding the use of *allInstances* in the second example is not so simple. It requires extending the original context diagram (in Figure 4.5) as shown in Figure 4.6. The class *Conference* has been newly introduced. An appropriate reformulation of the second constraint above could be

**context** Conference
**inv** self.submitted_papers ->forAll(p1, p2 |
p1 <> p2 implies p1.number <> p2.number)

Here is another elegant solution

**context** p1,p2:Papers
**inv** p1 <> p2 implies p1.number <> p2.number)

This is, at the moment, not legal OCL syntax, since the declaration of two

variables in the contect class is not supported.

The initial submission [Boldsoft *et al.*, 2002] to the UML 2.0 OCL Request for Proposals proposes to omit *OclType* alltogether. This proposal integrates a meta-model for OCL into the already existing meta-model of UML. Most of the operations for *OclType* then become superfluous, since they are inherited within the meta-model. This applies, in fact, for all operations except *allInstances*.

## 4.7 The iterate operation

### 4.7.1 Example

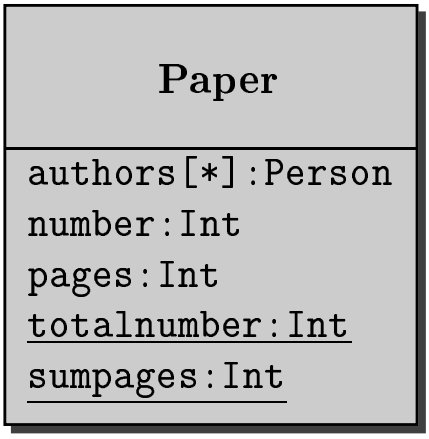



Figure 4.7: Context class for constraint with *iterate*

In the context of the class in Figure 4.7 the following constraint

**context** p:Papers
  **inv** Papers.allInstances -> iterate(x:Paper ; y:Int = 0 | y+x.pages)
    = Papers.totalnumber

expresses that fact that the class scope attribute *sumpages* is the sum of the number of pages taken over all papers.

### 4.7.2 Constraint Syntax

The general form of the *iterate* construct is shown in Figure 4.8 subject to the following restrictions:

1. $y$ ist different from $x$,
2. $t$ does not contain $y$,
3. $t_0$ does not contain $x$ nor $y$,

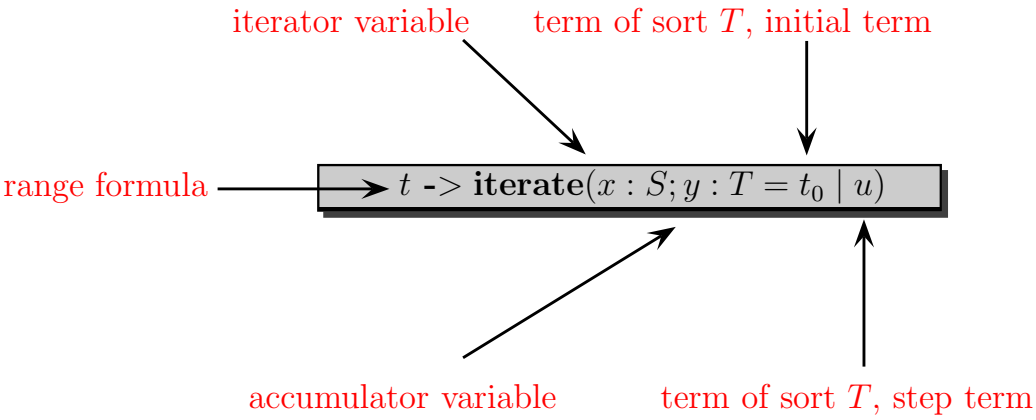



Figure 4.8: Syntax of the *iterate* construct

### 4.7.3 Meaning of the Constraint

Consider an expression $exp$ of the form $t$ -> **iterate**$(x; y = t_0 \mid u)$. We describe how to compute the meaning of $exp$, assuming that we know already how to evaluate the subexpressions $t$,$t_0$ and $u$. Subexpression $t$ evaluates to a set of objects, say $A = \{a_1, \ldots, a_n\}$ and $t_0$ to an object $m_0$. Subexpression $u$ will typically contain both variables $x$ and $y$. So, if we supply objects $a$ for $x$ and $m$ for $y$ then evaluation of $u$ will yield an object $m'$, we will write this as $m' = u(a, m)$. After this preliminaries we are ready for the evaluation of $exp$.

We start with $m_0$, as definied above. We continue with $m_{i+1} = u(a_{i+1}, m_i)$. Then $m_n$ is the value of *exp*.

This definition depends in general on the ordering of the elements $\{a_1, \ldots, a_n\}$. If $t$ is of type *sequence* then we, naturally, use the order given by the sequence. In the other cases the UML standard leaves the issue unresolved. It seems sensible to require that it is the user's responsibility to make sure, that he only uses the iterate construct on sets when its result does not depend on the order of the elements in the set.

### 4.7.4 Another Example

The built-in type *String* provides the operation *substring* to access the substring between positions *lower* and *upper* in given string, see Subsection 12.1.3. The are other operations on strings that one would wish to have available, e.g. does *string*2 occur in *string* as a substring? Or, even better, the set of all positions in *string*, wher an occurence of *string*2 begins. So let us add a new opperation *occurences*. Following the style used in the standard, see the Appendix 12 this leads to:

**string.occurences(string2:String):Set(Integer)** The set of positions in string where an occurence of string2 as a substring starts. Strings start with position 0.
**pre** : string2.size =< string.size
**post**: result = { 0 .. (string.size - string2.size) } -> iterate(x;y={} |
**if** string.substring(x,x+string2.size-1) = string2
**then** y -> including(x)
**else** y)

This easily allows us to introduce further useful operations, like

**string.substringOcc(string2:String):Boolean** True if string2 occurs at least once as a substring in string.
**post**: result = (string2.size =< string.size) and
not (string.occurences(string2) -> isEmpty)

### 4.7.5 Comment

Most of the operations on sequences, bags and sets can be reduced to an application of *iterate*.

The expression

$$t->forAll(x \mid a)$$

where $t$ is an expression of type $Set(T)$, $x$ is a variable of type $T$, and $a$ is an expression of type $Boolean$ can be equivalently expressed by

$$t->iterate(x; y : Boolean = true \mid y \text{ and } a)$$

where $y$ has been chosen to not appear in $a$.

Here, are some comments on the built-in type *String*.

The standard does not determine the starting position of strings. Here we assume that the first position is 0.

The specification of the substring operation is underdetermined. It does not require the obvious precondition lower $\leq$ upper, nor does is say what the result should be in case upper $<$ lower.

## 4.8 Collecting Elements

### 4.8.1 Example

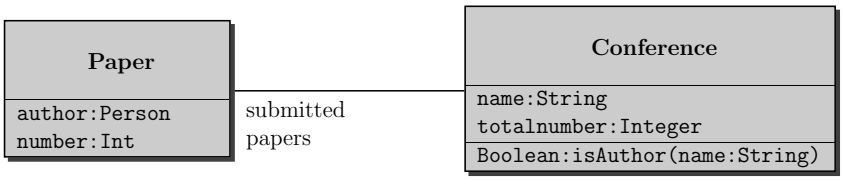



Figure 4.9: The *isAuthor* operation

The diagram shown in Figure 4.9 extends our running scenario by adding a new operation *isAuthor* to the conference class. This operation takes one

argument *name* of type *String* and returns a Boolean value. We want this value to be *true* if an author by the name *name* has submitted a paper.

**context** *c*:Conference::isAuthor(name:String)
**pre** true
**post** result = c.sp->collect(p | p.author.name)->includes(name)

Here we have used *sp* to abbreviate *submitted papers*.

This is our first example of an operation that returns a value, and also our first encoounter with the OCL key word *result*. This may be used in post conditions of operation constraints and refers to the returned value of the operation.

Also the *collect* operation is new. In the above example

c.sp->collect(p | p.author.name)

denotes the set of names of authors of submitted papers. Note, that we have for the start simplified things a bit, by assuming a unique author for each paper.

### 4.8.2 Constraint Syntax

The *collect* operation may be applied to collections *s* and has the general form

s->collect( var | expr )

where *expr* is an arbitrary OCL expression typically containing the variable *var*. If *var* is clear form the context the abbreviated form

s->collect( expr )

may be used.

### 4.8.3 Meaning of the Constraint

If *s* is a set, bag, or sequence then

s->collect( var | expr )

denotes the set, bag, or sequence of all elements *expr* when *var* is instantiated in turn with all elements in *s*.

### 4.8.4 Comment

The collect operation can be defined using the iterate operation:

set->collect(x |expr ) : Set(T) = set->iterate(x; acc : Set(T) = Set{} | acc->including(expr) )

where expr is an OCL expression of type T.

Let us go back to the example of this section and drop the simplification on the author attribute, i.e. we now use the attribute *authors* of type *Sequence*(*String*). Then

c.sp->collect(p | p.authors.name)

evaluates to the bag of all strings that appear as others of a submitted paper. Notice, that immediate implicit flattening occurs: we do not obtain a set (or bag) of sets of names, but one flattened set of names.

It has become quite customary to abbreviate OCL expressions of the kind s->collect( e | e.attribute) by s.attribute.

## 4.9 Selecting Elements

### 4.9.1 Example

Figure 4.10 contains a class operation *countShortPapers* that may be defined by the following constraint:

**context** Paper::countShortPapers():Integer
**pre** true
**post** result =
Paper.allInstances->select(p | p.pages < 10)->size

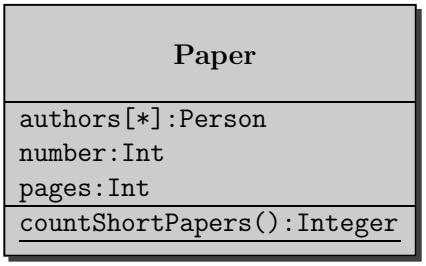



Figure 4.10: Context class for *select* Example

The select operation applied to the set *Paper.allInstances* produces the subset of all papers p from this set satisfying the condition p.pages $< 10$. Finally the size of this subset is determined by calling the built-in function *size* and returned as the result of the operation *countShortPapers()*.

### 4.9.2 Constraint Syntax

The syntax of the *select* operation parallels that of *collect*. It may be applied to collections $s$ and has the general form

s->select( var | expr )

where *expr* is an OCL expression of type *Boolean* typically containing the variable *var*. If *var* is clear form the context the abbreviated form

s->select( expr )

may be used.

### 4.9.3 Meaning of the Constraint

If $s$ is a set, bag, or sequence then

s->select( var | expr )

denotes the set, bag, or sequence of all elements $p$ from $s$ satisfying *expr*, i.e. such that *epxr* evaluates to *true*.

### 4.9.4 Comment

The *select* operation can be defined using the iterate operation:

s->select(x | expr ) : Set(T) = s->iterate(x; acc : Set(T) = Set{} |
if expr then acc->including(x)
else acc endif)

where $s$ is of type $Set(T)$ and *expr* is an OCL expression of type *Boolean*.

## 4.10 Quantifiers

### 4.10.1 Example

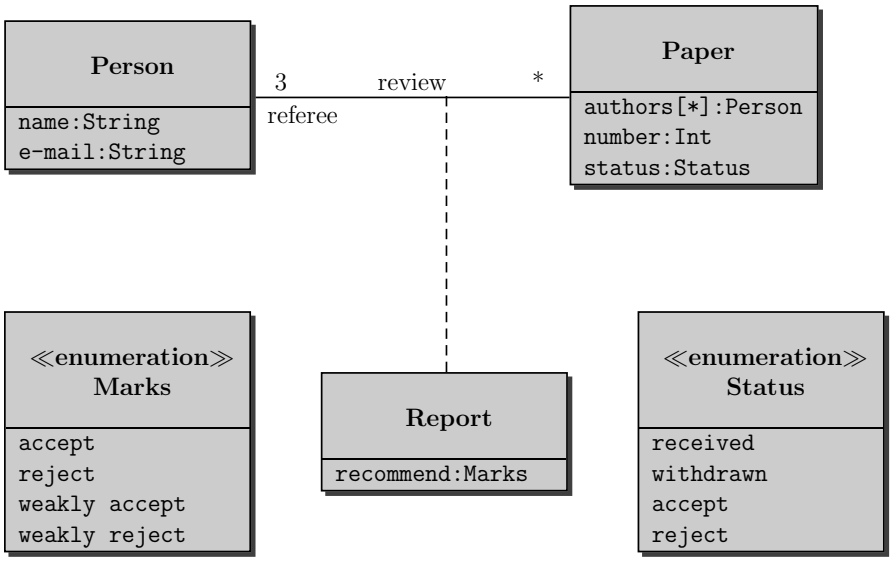



Figure 4.11: A context diagram for quantifiers

In the context of Figure 4.11 the following contraint may be formulated:

```
context p:Papers
   inv p.referee -> forAll( ref:Person | ref.Report.recommend = reject)
         implies p.status = reject
```

This constraint formulates the requirement that every paper, such that all referees recommend rejection should receive status *reject*. Notice, that the constraint effects navigation from a paper $p$ to the set of its referees, and from there to the association class *Report* and its attribute *recommend*. This is our first example of navigation into an association class.

### 4.10.2 Constraint Syntax

The *forAll* operation may be applied to collections $s$ and has the general form

```
s->forAll( var:Type | expr )
```

where *expr* is an OCL expression of type Boolean typically containing the variable *var*. If *var* is clear form the context the abbreviated form

```
s->forAll( expr )
```

may be used. A similar syntax applies for the *exists* operation:

```
s->exists( var:Type | expr )
```

### 4.10.3 Meaning of the Constraint

The meaning of the constraints

```
s->forAll( var:Type | expr )
s->exists( var:Type | expr )
```

is as one would expect. In a particular snapshot $D$ the Boolean expression *expr* is evaluated for every element $a$ in the set, bag, or sequence $s$. If the result is always true the *forAll* operation evaluates also to true. If at least once the value true occurs the *exists* operation evaluates to true. For empty collections *forall* is always true and *exists* is always false.

### 4.10.4 Comment

The *forAll* operation is usually called *universal quantification* and the *exists* operation *existential quantification*

Universal and existential quantification can be expressed using the *iterate* operation.

s->forAll(x | expr ) = s->iterate(x; acc : Boolean = true | acc and expr)

s->exists(x | expr ) = s->iterate(x; acc : Boolean = false | acc or expr)

## 4.11 Refering to previous values

### 4.11.1 Example


Figure 4.12: The operation *addPaper*

The diagram shown in Figure 4.12 extends our running scenario by adding a new operation *addPaper* to the conference class. Having the conference class at our disposal it makes more sense to record the number of submitted papers as an attribute to this class. At some previous diagrams we totalnumber was modeled as a class attribute of the class Paper, see Figures 4.7 or 4.1.

**context** *c*:Conference::addPaper()
**pre** true
**post** totalnumber = totalnumber@pre + 1

The OCL expression *totalnumber@pre* refers to the value of the attribute *totalnumber* before the evaluation of the operation *addPaper*. The expression *totalnumber* without suffix refers of course to the value of this attribute after execution of *addPaper*.

### 4.11.2 Constraint Syntax

The suffix @pre can only be used in postconditions. Then it can be attached to arbitrary attribute and association names. Multiple occurences of @pre in an expression are perfectly possible. Consider the diagram in Figure 4.13 We think of *People* as the class of customers of a bank. There is an attribute *pa* in *People* that associates with every costumer his personal assistant. Personal assistants as all other employees of the bank have a phone number. Assume the bank, or a branch of it, moves into a new building. Phone numbers may change and also the distribution of the customers among the personal assistance was reconsidered on this occasion. The method $m$ effects all these changes. For $c : People$ there are in OCL four possible expressions refering to phone numbers

| | |
|---|---|
| *c.pa.phone* | the new phone number of the current p.a. |
| *c.pa@pre.phone* | the new phone number of the previous p.a. |
| *c.pa.phone@pre* | the old phone number of the current p.a. |
| *c.pa@pre.phone@pre* | the old phone number of the previous p.a. |

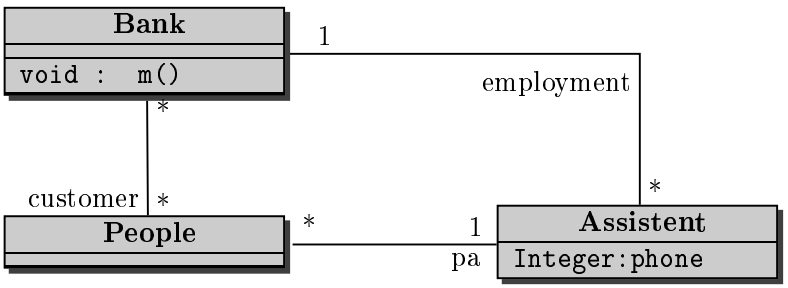



Figure 4.13: A scenario for multiple uses of @pre

### 4.11.3 Meaning of the Constraint

We have not seen operator constraints since Section 4.2. Then, we postponed the question, what in the meaning of an operation? Now, is the right place to come back to it. Of course, we all have some intuitive meaning what an operation should be. In particular, when we think of an implementation of a methods, e.g. as a Java method. But, at the level of an UML/OCL specification there is no implementation. All we have at this stage of program development is the declaration that there should be a method of a given signature. That is to say, it is specified which objects may call the method, what are the types of possible arguments, if any, what is the result type, if any. The only further information if given in the form of pre- and postconditions. For this to make sense, we need to know possible starting states of an operation and their corresponding end states. This leads us to the definition, that the meaning of an operation $m$ is a set $\rho(m)$ of pairs of states. If $(S_1, S_2)$ is a pair in $\rho(m)$ then $m$ called in $S_1$ will terminate in state $S_2$. This also encodes the possibility of non-termination: if for $S_1$ there is no pair in $\rho(m)$ with $S_1$ as its first component, then $m$ started in $S_1$ does not terminated. Furthermore, this definition also leaves open the issue of non-determinism. It is not exluded that $(S_1, S_2)$ and $(S_1, S_3)$ with $S_2 \neq S_3$ occur in $\rho(m)$. If one whishes to consider only deterministic operations, which will be the case for most of the time, then one has to stipulate that $\rho(m)$ is a partial function rather than an arbitrary relation, i.e. for all $(S_1, S_2)$ and $(S_1, S_3)$ in $\rho(m)$ the equality $S_2 = S_3$ follows.

### 4.11.4 Comment

## 4.12 Role Based Access Control

> One of the most challenging problems in managing large networked systems is the complexity of security administration. Today, security administration is costly and prone to error because administrators usually specify access control lists for each user on the system individually. Role based access control (RBAC) is a technology that is attracting increasing attention, particularly for commercial applications, because of its potential for reducing the

> complexity and cost of security administration in large networked applications.
>
> With RBAC, security is managed at a level that corresponds closely to the organization's structure. Each user is assigned one or more roles, and each role is assigned one or more privileges that are permitted to users in that role. Security administration with RBAC consists of determining the operations that must be executed by persons in particular jobs, and assigning employees to the proper roles. Complexities introduced by mutually exclusive roles or role hierarchies are handled by the RBAC software, making security administration easier.

This quote is taken from the RBAC web-page[1] of NIST, the National Institute of Standards and Technology. Besides maintaining this topical web-page NIST is also involved in standardization efforts in the area of RBAC, see [Ferraiolo *et al.*, 2000]. We will present in this section an UML/OCl model of the most important features of this standard, following its subdivision into the four parts as given by the captions of the following subsections. Concerning the class Permission we haven taken a slightly more abstract view than in the [Ferraiolo *et al.*, 2000] in that we treat permissions as an atomic concept and do not consider the association of objects and operations with permissions.

### 4.12.1 RBAC Core

Figure 4.14 shows the main classes and associations of the core model of rule based access control. The association *ua*, user assignment, establishes a relation between users and roles. A user may be assigned multiple roles and a role may be associated with more then one users. The same many-many-relationship also holds true for the association *pa*, permission assignment. On the other hand every session has a unique user associated via *sra*, session role assignment. But a user may run more than one session.

Note also the role names *assigned_users*, *assigned_permissions*, *user_sessions*, *session_roles* and *session_user*.

For *s*:*Session* we will use the abbreviation *s.avail_session_perms* for *s.session_roles.permission*.

[1] http://csrc.nist.gov/rbac/

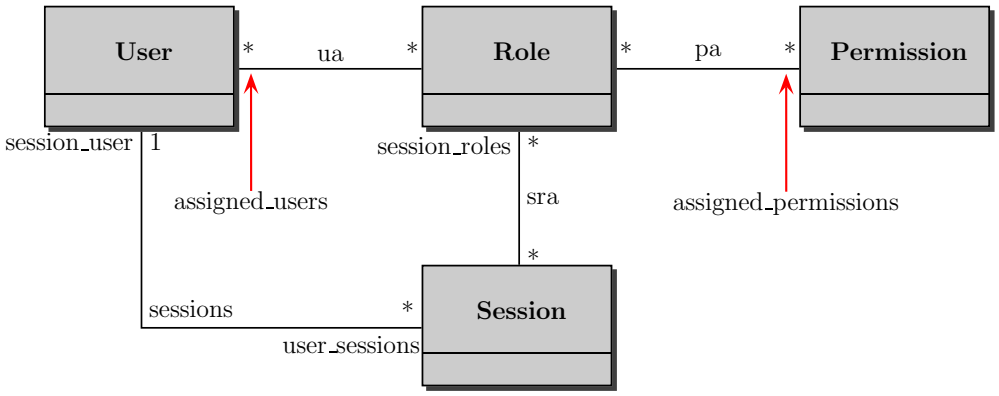



Figure 4.14: Class diagram for RBAC core

The only OCL invariant constraint added to Figure 4.14 at this level of the model is

**context** u:User
**inv** u.role->includesAll(u.sessions.session_roles)

Figure 4.15 shows the attributes and operations on class *User*. There is only one, rather simple, attribute, the name of a user. The intended meaning of the operations is described by operation constraints. The following OCL constraints are basically translations from the formal description in [Ferraiolo *et al.*, 2000] using Z.

**context** *u*:User::assignUser(r:Role)
**pre** not(r.assigned_users->includes(u))
**post** r.assigned_users = r.assigned_users@pre->including(u)

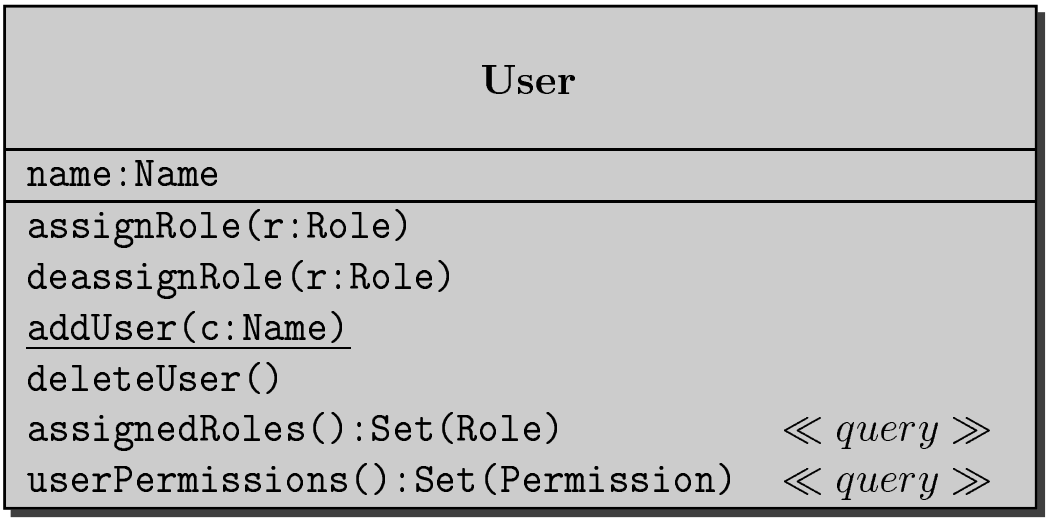



Figure 4.15: The class User

**context** $u$:User::deassignUser(r:Role)
**pre** r.assigned_users->includes(u)
**post** r.assigned_users = r.assigned_users@pre->excluding(u) **and**
u.user_sessions =
u.user_sessions@pre->reject(s : s.session_roles->includes(r))

**context** $u$:User::addUser(c:Name)
**pre** User->forAll($u_1$ : $u_1$.name <> c)
**post** User->exists( $u_1$ | User@pre->excludes( $u_1$)
**and** $u_1$.name = c **and**
User = User@pre->including($u_1$))

We have used a shortcut notation here. User->forAll(...) is not legal OCL syntax. It should correctly read User.allInstances->forAll(...). We will use this shortcut consistently in the following: If $C$ is a class from the context diagram then $C$-> will be shorthand for $C$.allInstances->.

One would have liked to write the last constaint as

**context** $u$:User::addUser(c:Name)
**pre** User->forAll(u : u.name <> c)
**post** User = User@pre->includes(new) **and**
new.name = c

But, OCL does **not** provide for this. There is however a shorthand notation for User@pre->excludes($u_1$) namely $u_1$.oclIsNew(). Thus the above constraint could also be written as:

**context** $u$:User::addUser(c:Name)
**pre** User->forAll($u_1$ : $u_1$.name <> c)
**post** User->exists( $u_1$ | $u_1$.oclIsNew
**and** $u_1$.name = c **and**
User = User@pre->including($u_1$))

**context** $u$:User::deleteUser()
**pre** true
**post** User = User@pre->excluding(u) **and**
Session =
Session@pre->reject(s : u.user_sessions->includes(s))

**context** $u$:User::assignedRoles()
**pre** true
**post** result = u.role

**context** $u$:User::userPermissions()
**pre** true
**post** result = u.role.permission

Attributes and operations for the class Role can be seen in Figure 4.16. Here are their definitions via pre- and postconditions.

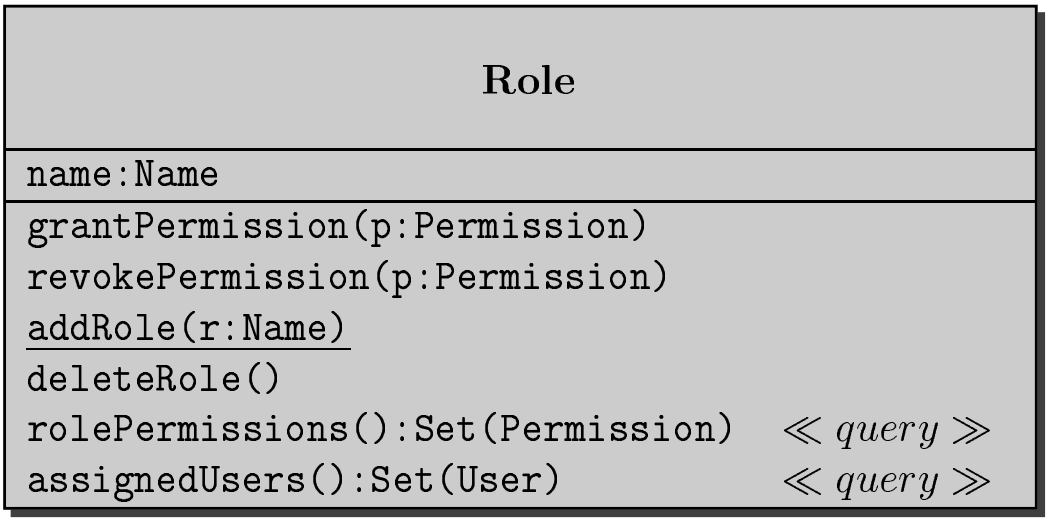



Figure 4.16: The class Role

**context** $r$:Role::grantPermission(p:Permission)
**pre** true
**post** r.assigned_permissions =
r.assigned_permissions@pre->including(p)

**context** $r$:Role::revokePermission(p:Permission)
**pre** r.assigned_permissions->includes(p)
**post** r.assigned_permissions =
r.assigned_permissions@pre->excluding(p)

**context** *r*:Role::addRole(c:Name)
**pre** Role->forAll(r : r.name <> c)
**post** Role->exists( $r_1$ | Role@pre->excludes( $r_1$)
**and** $r_1$.name = c **and**
Role->forAll( w |
Role@pre->includes(w) **or** w = $r_1$))

**context** *r*:Role::deleteRole()
**pre** true
**post** Role = Role@pre->excluding(r) **and**
Session =
Session@pre->reject(s : s.session_roles->includes(r))

**context** *r*:Role::rolePermissions()
**pre** true
**post** result = r.assigned_permissions

**context** *r*:Role::assignedUsers()
**pre** true
**post** result = r.assigned_users

**context** *s*:Session::addActiveRole(u:User,r:Role)
**pre** r.assigned_users->includes(u) **and**
s.session_user = u **and not** (s.session_roles->includes(r))
**post** s.session_roles = s.session_roles@pre->including(r)

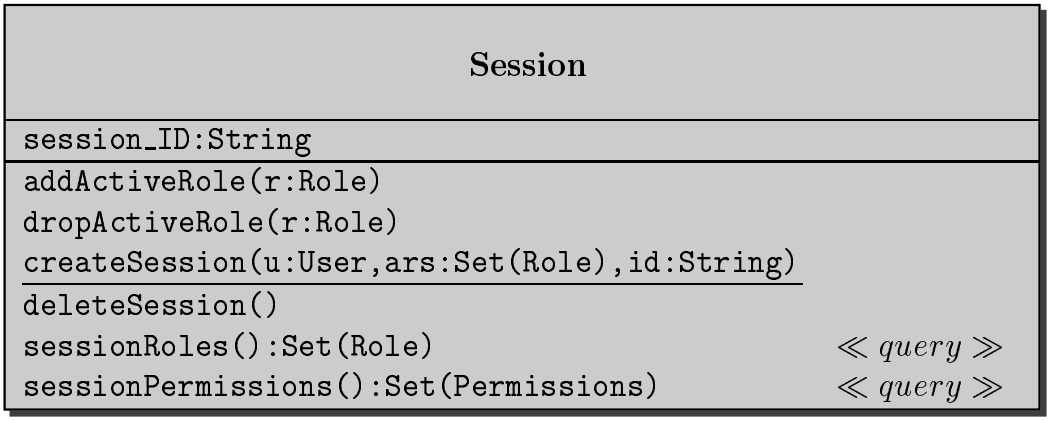



Figure 4.17: The class Session

**context** *s*:Session::dropActiveRole(u:User,r:Role)
**pre** r.assigned_users->includes(u) **and**
s.session_user = u **and** s.session_roles->includes(r)
**post** s.session_roles = s.session_roles@pre->excluding(r)

**context** *s*:Session::createSession(u:User,ars:**Set**(Role),id:String)
**pre** Sessions->**forAll**(s : s.session_ID <> id) **and**
u.au->includesAll(ars)
**post** u.user_sessions->exists( $s_1$ | u.user_sessions@pre->excludes( $s_1$)
**and** $s_1$.session_ID = id **and**
u.user_sessions->forAll( w |
u.user_sessions@pre->includes(w) **or** w = $s_1$)) **and**
s.session_roles = ars

Here, the variable name *ars* refers to *active role set*.

**context** *s*:Session::deleteSession(u:User)
**pre** s.session_user = u
**post** Session = Session@pre->excluding(s) **and**
u.user_sessions = u.user_sessions@pre->excluding(s)

**context** *s*:Session::sessionRoles()
**pre** true
**post** result = s.session_roles

**context** *s*:Session::sessionPermissions()
**pre** true
**post** result = s.session_roles.assigned_permissions

**context** *s*:Session::checkAccess(op:Operation,obj:Object)
**pre** true
**post** result = s.session_roles.assigned_permissions->
exists(p | p.operation = op **and** p.object = obj)

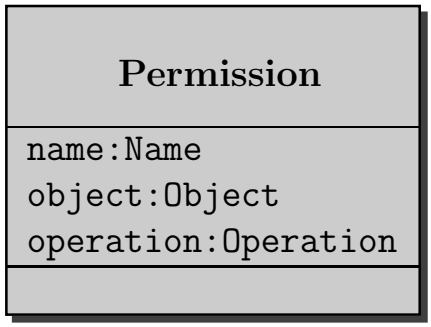



Figure 4.18: The class Permission

Figure 4.18 shows the last class to be considered for the RBAC core. The class Permission does not have any operations or methods. The attributes

use the classes *Objects* and *Operations* that are assumed to be available. The standard [Ferraiolo *et al.*, 2000] states that *Permission* is a subset of the cartesian product of *Object* and *Operation*. The mathematical content of this stipulation is covered by the following constraint:

**context** $p_1$, $p_2$:Permission
**inv** ($p_1$.object = $p_2$.object **and** $p_1$.operation = $p_2$.operation)
**implies** $p_1 = p_2$

### 4.12.2 Hierarchical RBAC

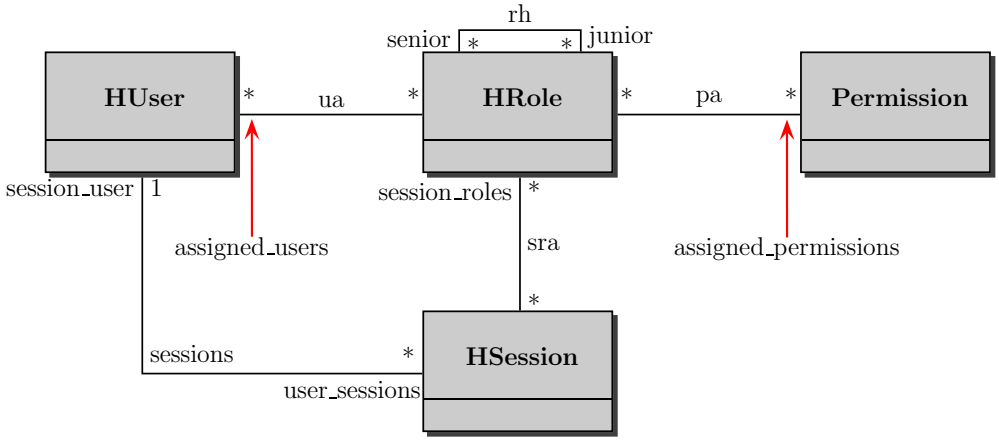



Figure 4.19: Class diagram for RBAC with hierarchy

The hierarchical role-based access model is obtained by adding one further association to the core model, named $rh$ for role hierarchy in Figure 4.19. It establishes a relation between a junior role $r_1$ and a senior role $r_2$, which we denote by $r_1 \preccurlyeq_{rh} r_2$. (In [Ferraiolo *et al.*, 2000] the notation $r_1 \nu r_2$ is used for $r_1 \preccurlyeq_{rh} r_2$.) The transitive closure of $\preccurlyeq_{rh}$ is denoted by $<_{rh}$ (In [Ferraiolo *et al.*, 2000] $r_1 o\ r_2$ is used.) Also the classes are changed from Role to HRole, User to HUser and Session to HSession. The class Permission remains unchanged.

In the following we will make use of the following abbreviations

1. r.senior$^+$ := HRole->
   **iter**($r_1$ ; y:Set(HRole) = r.senior | y->union(y.senior))
2. r.senior* := HRole->
   **iter**($r_1$ ; y:Set(HRole) = { r } | y->union(y.senior))

Notice the subtle difference between these two definitions: r.senior* contains all elements of r.senior$^+$ and in addition also the role r itself.

Also notice the at first glance suprising fact that the iterator variable $r_1$ does not occur in the step formula. What is the effect of the step term? Well, it adds to the accumulator $y$ all roles that are one level above of at least one role in $y$. How often do we want to repeat this? As long as new elements are still added. It would be very complicated to add a condition to the effect that the addition of new roles is repeated till the set $y$ does not grow anymore. On the other hand it is save to say that the number of all elements in HRole is an upper bound on the number of necessary iterations. This is the only function of the iterator $r_1$.

Two types of hierachies are considered:

1. General role hierachies, where $<_{rh}$ is only assumed to be a partial order,
2. Limited role hierachies, where $<_{rh}$ is assumed to be a tree ordering, i.e. multiple inheritance is not allowed.

We formulate both constraints as invariants of the class *HRole*.

**context** r,$r_1$:HRole
  **inv** GeneralRH :
    r.senior-> **includes** $r_1$ **implies**
    **not** (HRole->exists($r_2$ |
      r.senior$^+$-> **includes** $r_2$ **and** $r_2$.senior$^+$-> **includes** $r_1$))
    **and**
    r.senior$^+$->exludes(r)

As usual y.senior is shorthand for y->**collect**(s | s.senior). GeneralRH is the name of the invariant.

The first part of the invariant *GeneralRH* says that the relation r.senior-> **includes** $r_1$ (in *pretty print* written as $r \lessdot r_1$) is an immediate successor relation with the more general relation $<_{rh}$. In ordinary mathematical notation this requirement reads $\forall r, r_1(r \lessdot r_1 \rightarrow \neg\exists r_2(r <_{rh} r_2 \wedge r_2 <_{rh} r_1)$

The second conjunction of *GeneralRH* exculdes cylces in $<_{rh}$.

In the limited role hierarchy multiple inheritance of roles is not allowed, this is to say a role cannot have two seniors.

**context** r,$r_1$,$r_2$:HRole
**inv** LimitedRH : GeneralRH **and**
(r.senior->includes($r_1$) **and** r.senior->includes($r_2$))
**implies** $r_1 = r_2$

Alternatively:

**context** r:HRole
**inv** LimitedRH : GeneralRH **and**
r.senior->size()≤1

Of course, we could have obtained the same effect by changing in Figure 4.19 the * multiplicity at the *senior end* of the association *rh* to 1.

In the UML model of hierarchical role-based access control also new operations will be added to the classes. Figure 4.20 shows the *HRole* class which arises from the class *Role* by adding four new operations. Instead of repeating every attribute and operation from the previously encountered class *Role* we introduce the shorthand notation shown in Figure 4.21.

**context** *r*:HRole::addInheritance($r_1$:HRole)
**pre** r <> $r_1$ **and not** (r.senior*->including($r_1$))
**and not** ($r_1$.senior*->including(r))
**post** r.senior->including($r_1$) **and**
HRole->forAll($r_2$,$r_3$ | (($r_2$ <> r **or** ($r_3$ <> r) **implies**
($r_2$.senior->includes($r_3$) **iff** $r_2$.senior@pre->includes($r_3$))

Here senior* is as above and A **iff** B is an abbreviation of (A **implies** B) **and** (B **implies** A)

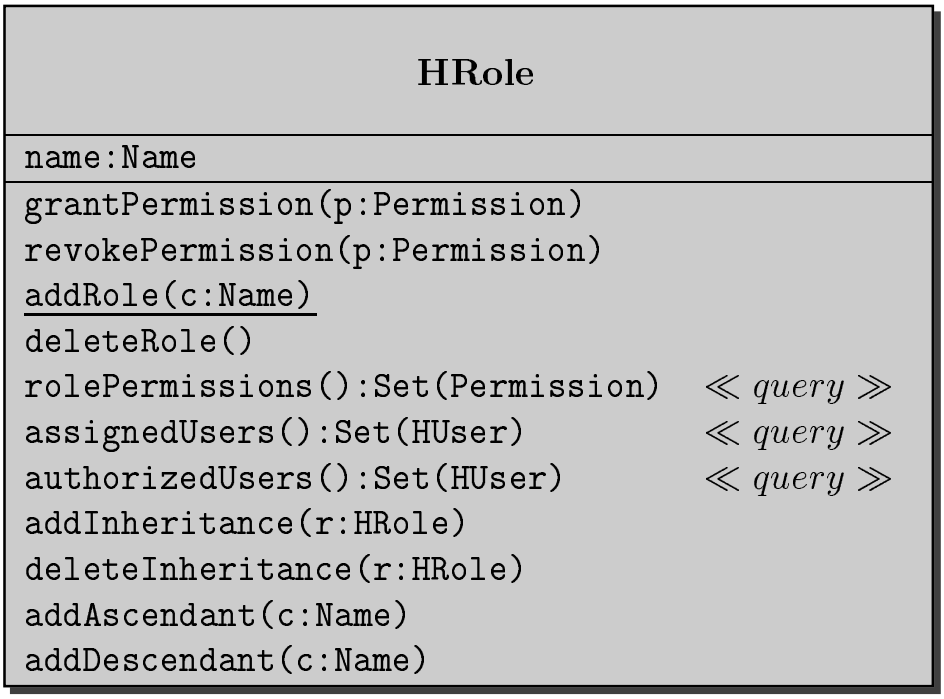



Figure 4.20: Class HRole

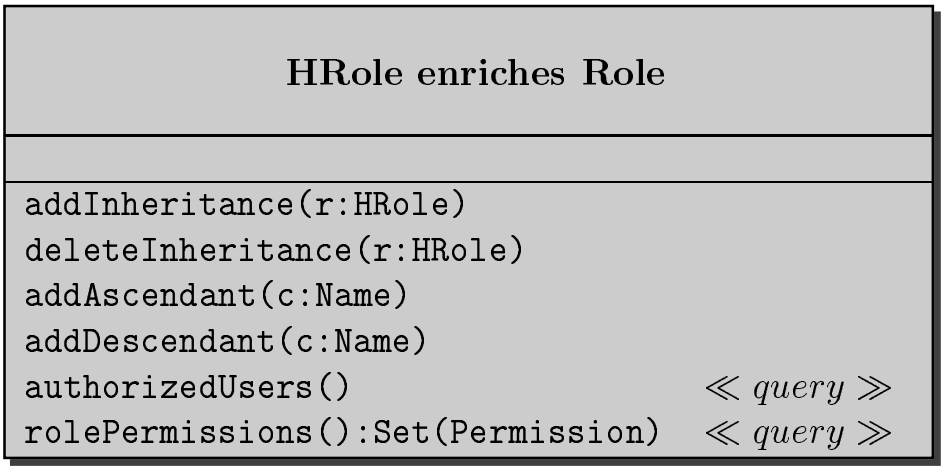



Figure 4.21: Shorthand for Class HRole

Using usual mathematical notation, as is done to a considerable extend in the specification language **Z** one could write $r_1 <_{rh} r$ instead of the OCL expression r$_1$.senior*->including(r). The above constraint would then read:

**context** $r$:HRole::addInheritance(r$_1$:HRole)
**pre** r <> r$_1$ **and** $r \not<_{rh} r_1$ **and** $r_1 \not<_{rh} r$
**post** $r <_{rh} r_1$ **and**
HRole->forAll(r$_2$,r$_3$ | ((r$_2$ <> r) **or** (r$_3$ <> r)) **implies**
($r_2 <_{rh} r_3$ **iff** $r_2 <_{rh}$ @*pre* $r_3$))

**context** $r$:HRole::deleteInheritance(r$_1$:HRole)
**pre** r.senior->includes(r$_1$)
**post** r.senior->excludes(r$_1$) **and**
HRole->forAll(r$_2$,r$_3$ | ((r$_2$ <> r **or** (r$_3$ <> r) **implies**
r$_2$.senior->includes(r$_3$) **iff** r$_2$.senior@pre->includes(r$_3$)

**context** $r$:HRole::addAscendant(c:Name)
**pre** HRole->forAll(r : r.name <> c)
**post** HRole->exists( r$_1$ | HRole@pre->excludes( r$_1$)
**and** r$_1$.name = c **and**
r.senior->includes( r$_1$) **and**
HRole = HRole@pre->including(r$_1$))

**context** $r$:HRole::addDescendant(c:Name)
**pre** HRole->forAll(r : r.name <> c)
**post** HRole->exists( r$_1$ | HRole@pre->excludes( r$_1$)
**and** r$_1$.name = c **and**
r$_1$.senior->includes( r) **and**
HRole = HRole@pre->including(r$_1$))

**context** $r$:HRole::authorizedUsers():Set(HUser)
**pre** true
**post** result =
r.senior*.assigned_users

Here senior* is an abbreviation for

r.senior* := HRole->
**iter**($r_1$ ; y:Set(HRole) = r.senior | y->union(y.senior))

This constraint reflects the assumption that a user that has a a role $r$ may also act in the roles $r_1$ with $r_1 \leq_{rh} r$. If we start with a role $r$ and ask what are the assigned users, then we collect all users assigned to $r$ itself but also all users that are assigned to roles $r_2$ with $r \leq_{rh} r_2$.

Finally, we come to consider the query operation rolePermissions(), which was already part of the class Role, but has nevertheless been listed again in the HRole definition. The reason is, that its meaning is redefined in HRole:

**context** $r$:HRole::rolePermissions()
**pre** true
**post** result = r.junior*.assigned_permissions

This covers the new class HRole. Now we turm to the modified class HUser.

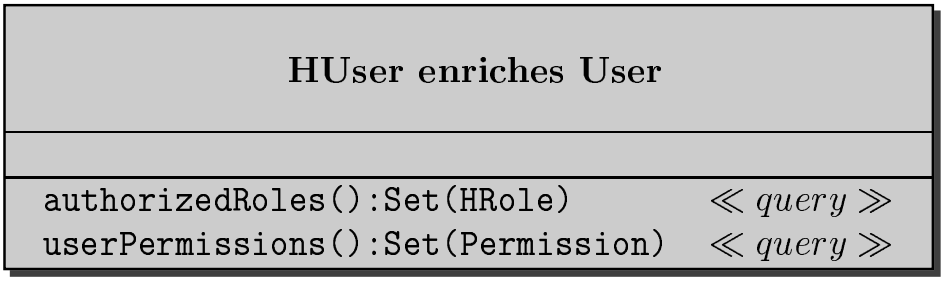



Figure 4.22: The Class HUser

**context** $u$:HUser::authorizedRoles():Set(HRole)
**pre** true
**post** result =
HRole->select( r | r.senior*.assignedUsers->includes(u))

The query operation userPermissions() already occured in the class User. It is listed again in the class HUser in Figure 4.22 since it is redefined. The wisdom of this decision could be debated. An alternative would have been to retain the operation userPermissions() unchanged and add a new one, called e.g. authorizedUserPermissions(). But, we stick as closely as possible to the proposal in [Ferraiolo *et al.*, 2000].

**context** $u$:HUser::userPermissions():Set(Permission)
**pre** true
**post** result =
u.authorizedRoles.permission

Finally we turn to the modified class HSession:

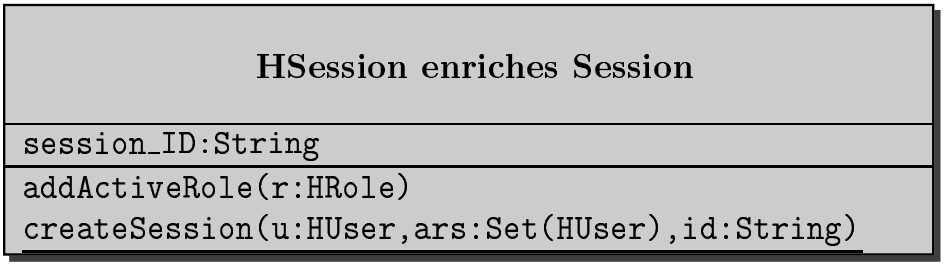



Figure 4.23: The Class HSession

**context** $s$:HSession::addActiveRole(u:HUser,r:HRole)
**pre** r.authorizedUsers->includes(u) **and**
s.session_user = u **and not** (s.session_roles->includes(r))
**post** s.session_roles = s.session_roles@pre->including(r)

The only difference to the constraint on addActiveRole in class Session is, that assigned_users has been replaced by authorizedUsers.

**context** *s*:HSession::createSession(u:HUser, ars:**Set**(HRole),id:String)
**pre** HSessions->**forAll**(s : s.session_ID <> id) **and**
u.authorizedRoles->includesAll(ars)
**post** u.user_sessions->exists( $s_1$ | u.user_sessions@pre->excludes( $s_1$)
**and** $s_1$.session_ID = id **and**
u.user_sessions->forAll( w |
u.user_sessions@pre->includes(w) **or** w = $s_1$)) **and**
s.session_roles = ars

The only difference is that u.au has been replaced by u.authorizedRoles.

As mention above, the class Permission remains unchanged. This is true in our case, since we did not really look inside it. In general, there will be operations or attributes in Permission. Even if those are not redefined, it might be necessary to change the type declarations, e.g. from User to HUser.

**Lemma 1** *From the constraints given above the following two invariants can be derived:*

*1.* **context** $r_1$, $r_2$*:HRole*
**inv** *UserInheritance:*
$r_1$*.senior*->includes(*$r_2$*) implies*
$r_1$*.authorizedUsers()->includesAll(*$r_2$*.authorizedUsers())*

*2.* **context** *r:HRole*
**inv** *PermissionInheritance:*
$r_1$*.senior*->includes(*$r_2$*) implies*
$r_2$*.rolePermissions()->includesAll(*$r_1$*.rolePermissions())*

Using $r <_{rh} s$ for r.senior*->includes(s) and concrete syntax of set theory these constraints can slao be presented as:

**context** $r_1$, $r_2$:HRole
**inv** UserInheritance:
$r_1 <_{rh} r_2$ implies
$r_2$.authorizedUsers() $\subseteq$ $r_1$.authorizedUsers()

**context** r:HRole
**inv** PermissionInheritance:
$r_1 <_{rh} r_2$ implies
$r_2$.rolePermissions() $\subseteq$ $r_2$.rolePermissions()

**Proof:**

### 4.12.3 Static Separation of Duty Relations

### 4.12.4 Dynamic Separation of Duty Relations

## 4.13 Exercises

**Exercise 4.13.1** *Is* $Set(Integer) \ll Collection(Real)$ *true?*

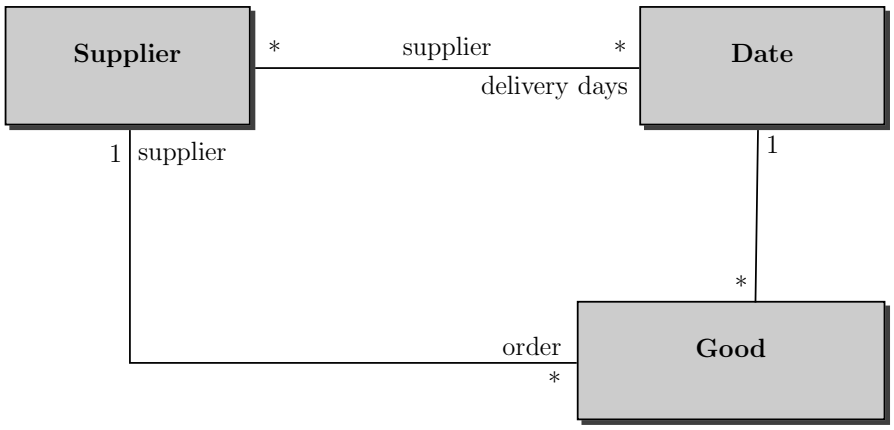



Figure 4.24: Scenario from Excerise 4.13.2

**Exercise 4.13.2** *Consider the scenario in Figure 4.24. A shopping center holds contracts with a number of suppliers in which the weekly orders of goods are agreed upton. There is also a fixed arrangement on the day(s) of the week a supplier delivers and a schedule what goods are delivered at each delivery date.*

1. *Formulate an OCL constraint that states that the order for each supplier is to be understood as the total weekly order.*

2. *Formulate an OCL constraint that states that the order is to be understood as order per delivery.*

# Chapter 5

# Systematic Introduction to OCL

po In this chapter we will give a complete description of the syntax and semantics of OCL expressions and contraints. Having read the definition by examples in Chapter 4 the reader should now be ready to understand a more concise definition. The first part will contain a rigorous mathematical definition of syntax and semantics of OCL. In a asecond part we will investigate syntax and semantics definition using UML/OCL itself. It is unavoidable that we repeat here some material that has already been covered in the previous chapter.

We try to stick as close as possible to the draft standard [Boldsoft *et al.*, 2002]

## 5.1 Vocabulary

### 5.1.1 A Bird's Eye View

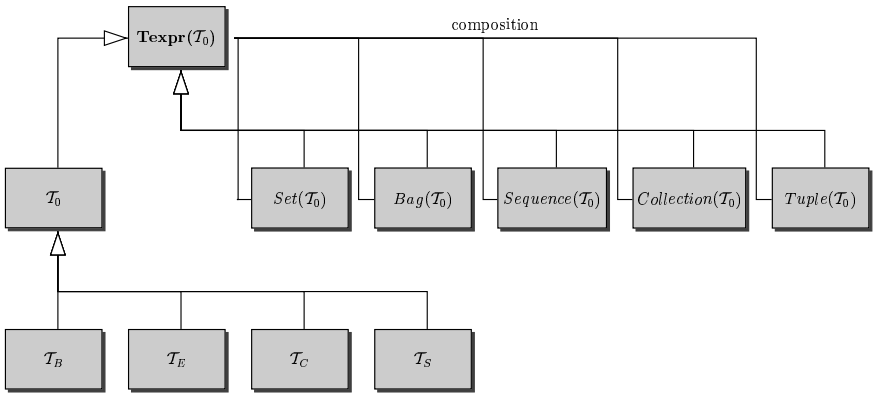



Figure 5.1: Top Level of Type Hierachy

A vocabulary $\Sigma = (\mathcal{T}, \Omega, <)$ for OCL expressions consists of the following parts

1. A set $\mathcal{T}$ of types,

2. A set $\Omega$ of operations,

3. A type hierarchy $<$.

Every operation $f \in \Omega$ is endowed with typing information, written as $f : t_1 \times, \ldots, \times t_n \rightarrow t_0$. with $t_i \in \mathcal{T}$. We call $t_0$ the result type and $t_1, \ldots, t_n$ the argument types. The typing information for $f$ is usually referred to as the signature of $f$.

An operation $c$ with signature $c \ : \rightarrow t$is usually called a *constant* or more precisely a constant symbol.

Furthermore $<$ is a binary relation on the set $\mathcal{T}$.

Some parts of the vocabulary depend on a fixed UML class diagram $\mathcal{D}$. In contexts where this dependence is crucial we will write $\Sigma^{\mathcal{D}} = (\mathcal{T}^{\mathcal{D}}, \Omega^{\mathcal{D}}, <^{\mathcal{D}})$

At the next level of detail the sets $\mathcal{T}$ and $\Omega$ are brocken down into disjoint subsets (see Figure 5.1):

$$\begin{array}{lcl} \mathcal{T} & = & Texpr(\mathcal{T}_0) \\ \mathcal{T}_0 & = & \mathcal{T}_B \cup \mathcal{T}_E \cup \mathcal{T}_C \cup \mathcal{T}_S \\ \Omega & = & \Omega_B \cup \Omega_E \cup \Omega_C \cup \Omega_S \cup \Omega_{Coll} \end{array}$$

1. $\mathcal{T}_B$ set of symbols for basic types,

2. $\mathcal{T}_E$ set of symbols for enumeration types,

3. $\mathcal{T}_C$ set of symbols for object types (also known as model types),

4. $\mathcal{T}_S$ set of symbols for special types.

5. $Texpr(\mathcal{T}_0)$ is the set of symbols for composite types constructed from $\mathcal{T}_0$.

### 5.1.2 Basic Types and Operations

$$\mathcal{T}_B = \{Integer, Real, Boolean, String\}$$

The operations $\Omega_B$ are listed in Appendix 12 or in the draft standard [Boldsoft *et al.*, 2002, Section 6.4].

Within $\mathcal{T}_B$ the only generalisation relation is

$$Integer \ < \ Real$$

### 5.1.3 Enumeration Types

The set of enumeration types $\mathcal{T}_E$ is completely determined by the UML class diagram $\mathcal{D}$.

$$\mathcal{T}_E = \{C \mid C \text{ is class in } \mathcal{D} \text{ with stereotype } \textit{enumeration}\}$$

$\Omega_E$ is the set of all enumeration literals occuring in the enumeration classes in $\mathcal{T}_E$. If $lit$ is an enumeration literal in $C$ then $lit$ has signature $lit :\rightarrow C$, i.e. literals are treated as operations without arguments.

### 5.1.4 Object Types

The set of object types is again completely determined by the UML class diagram $\mathcal{D}$. With every class $C$ in $\mathcal{D}$ we associate a unique type name $t_C$. In this text we will as a rule use for $t_C$ the name of $C$.

$$\mathcal{T}_C = \{t_C \mid C \text{ a class in } \mathcal{D}\}$$

Also the hierarchy relation $<$ on $\mathcal{T}_C$ is determined by $\mathcal{D}$.

The operations $\Omega_C$ come in four parts

1. Attribute operations
2. Query operations
3. Association operations
4. Predefined operations

**Attribute operations** For every attribute $a$ in class $C$ from $\mathcal{D}$ with value type $C_1$ there is an operation $a \in \Omega_C$ with signature

$$a : t_C \rightarrow t_{C_1}$$

If $a$ is a static attribute then its signature is

$$a : \rightarrow t_{C_1}$$

**Query operations** For every operation $op$ in class $C$ of diagram $\mathcal{D}$ which is stereotyped *isQuery* there is an operation $op \in \Omega_C$ with signature

$$op : t_C \times t_1 \times \ldots \times t_n \rightarrow t_{C_1}$$

where $t_i \in \mathcal{T}$, $t_C, t_{C_1} \in \mathcal{T}_C$ with $C_1$ the result type of $op$ and $t_1, \ldots, t_n$ the types of the arguments of $op$.

If $op$ is a query with class scope (static query) then the argument type $t_C$ is dropped in the above signature.

**Association operations** For the purpose of this definition we consider only binary associations. It should be obvious how to extend it to associations among more than two classes.

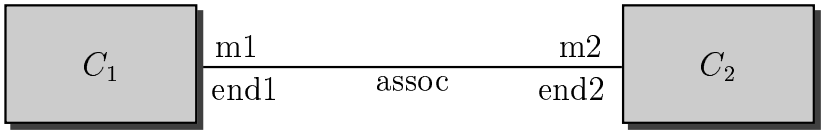



Figure 5.2: Top level meta model of OCL expressions

For every association $assoc$ in the diagram $\mathcal{D}$ with the notation from Figure 5.2 there will be two operations $f_{12}$ and $f_{21}$ in $\Omega_C$. We will use the name of the association end $end2$ as a name for $f_{12}$, if $end2$ is given in $\mathcal{D}$, otherwise we use the name of class $C_2$ starting with a lower case letter. To name $f_{21}$ we use $end1$ if given and the name of class $C_1$ otherwise. The signature of $f_{12}$ will be

$$f_{12} : t_{C_1} \rightarrow \ target\_type_2$$

Here $target\_type_2$ depends on the multiplicity $m2$.

$$target\_type_2 = \begin{cases} C_2 & \text{if} \quad m2 = 1 \\ Set(C_2) & \text{if} \quad m2 \neq 1 \\ Sequence(C_2) & \text{if} \quad m2 \neq 1 \text{ and } end2 \\ & \quad \text{is decorated by } ordered \end{cases}$$

The OCL standard sets the case $m2 = \{0, 1\}$ apart and proposes the signature

$$f_{12} : t_{C_1} \to C_2$$

in this case, with the understanding that $f_{12}$ is a partial function, i.e. could be not defined for some argument values. We strongly discourage this practice.

The signature of $f_{21}$ is

$$f_{21} : t_{C_2} \to \ target\_type_1$$

where $target\_type_1$ depends in the same way on $m1$.

**Predefined operations** For every class $C$ in diagram $\mathcal{D}$ $\Omega_C$ contains a operation symbol $allinstances_C$ of type $Set(t_C)$.

### 5.1.5 Collection and Tupel Types

The set $Texpr(\mathcal{T}_0)$ is defined by

1. If $t \in \mathcal{T}_0$ then $t \in Texpr(\mathcal{T}_0)$.
2. If $t \in \mathcal{T}_0$ then
   (a) $Set(t) \in Texpr(\mathcal{T}_0)$.
   (b) $Sequence(t) \in Texpr(\mathcal{T}_0)$.
   (c) $Bag(t) \in Texpr(\mathcal{T}_0)$.
   (d) $Collection(t) \in Texpr(\mathcal{T}_0)$.
3. If $t_1, \ldots, t_n \in \mathcal{T}_0$ and $l_1, \ldots, l_n$ are names for labels then $Tuple(l_1 : t_1, \ldots, l_n : t_n) \in Texpr(\mathcal{T}_0)$.

**Nested version** The new standard allows to nest type constructors. Thus $Texpr^{nest}(\mathcal{T}_0)$ is defined by

1. If $t \in \mathcal{T}_0$ then $t \in Texpr^{nest}(\mathcal{T}_0)$.
2. If $Texpr^{nest}(\mathcal{T}_0)$ then

(a) $Set(t) \in Texpr^{nest}(\mathcal{T}_0)$.

(b) $Sequence(t) \in Texpr^{nest}(\mathcal{T}_0)$.

(c) $Bag(t) \in Texpr^{nest}(\mathcal{T}_0)$.

(d) $Collection(t) \in Texpr^{nest}(\mathcal{T}_0)$.

3. If $t_1, \ldots, t_n \in Texpr^{nest}(\mathcal{T}_0)$ and $l_1, \ldots, l_n$ are names for labels then $Tuple(l_1 : t_1, \ldots, l_n : t_n) \in Texpr^{nest}(\mathcal{T}_0)$.

We will in our description stick with the old, unnested version.

For every applicable type $t$ the following generalisation relations are in force

$$Set(t) < Collection(t), Bag(t) < Collection(t), Sequence(t) < Collection(t)$$

The operations in $\Omega_{coll}$ are listed in Appendix 12. This operations allow you to manipulate sets.

Here are additional operations, called *constructors*, that create sets.

For every class $t \in \mathcal{T}_0$ and every natural number $n \geq 0$ the following operations are in $\Omega_{coll}$:

$$\begin{array}{lcl} mkSet_t^n & : & t \times t \times \ldots \times t \ \rightarrow \ Set(t) \\ mkBag_t^n & : & t \times t \times \ldots \times t \ \rightarrow \ Bag(t) \\ mkSequence_t^n & : & t \times t \times \ldots \times t \ \rightarrow \ Sequence(t) \end{array}$$

### 5.1.6 Special Types and Operations

There are just three special types

$$\mathcal{T}_S = \{OCLAny, OCLState, OCLVoid\}$$

and $\Omega_S = \{\bot\}$, where $\bot$ is a constant symbol of type $OCLVoid$, i.e. has signature

$$\bot : \rightarrow OCLVoid$$

$OCLAny$ is a supertype of all types and $OCLVoid$ is a subtype of all types in $\mathcal{T}_B \cup \mathcal{T}_E \cup \mathcal{T}_C$.

### 5.1.7 Type Hierarchy

Let us first collect all generalisation relations that have been stated scattered over the last subsections:

1. $Integer < Real$,
2. $t_{C_1} < t_{C_2}$ if in the diagram $\mathcal{D}$ $C_1$ is declared a subclass of $C_2$,
3. $Set(t), Bag(t), Sequence(t) < Collection(t)$,
4. If $t_1 < t_2$ then $Set(t_1) < Set(t_2)$, $Bag(t_1) < Bag(t_2)$, $Sequence(t_1) < Sequence(t_2)$, $Collection(t_1) < Collection(t_2)$,
5. If $t_1 < t'_1, \ldots, t_n < t'_n$ then $Tuple(l_1 : t_1, \ldots, l_n : t_n) < Tuple(l_1 : t'_1, \ldots, l'_n : t_n)$
6. $t < OCLAny$, $OCLVoid < t$ for all $t \in \mathcal{T}_B \cup \mathcal{T}_E \cup \mathcal{T}_C$.

The relation $\prec$ is the transitive closure of the hierarchy relation$<$, i.e. the least relation in $\mathcal{T}$ satisfying

1. If $t_1 < t_2$ then $t_1 \prec t_2$.
2. $\prec$ is transitive.

As usual we use $t \preceq s$ to stand for $t \prec s$ or $t = s$. $t \preceq s$ is read as $t$ conforms to $s$.

## 5.2 Syntax of OCL Expressions

Let $\Sigma = (\mathcal{T}, \Omega, <)$ be a fixed vocabulary. The following definition are all understood with reference to $\Sigma$: If necessary we will write more precisely $Expr_t(\Sigma)$ instead of $Expr_t$. Besides the symbols from $\Sigma$ we assume that there an infinite set $Var_t$ of variables available for every type $t$. It will sometimes be convenient to have a notation $Var$ for the set of all variables, i.e. $Var = \bigcup_{t \in \mathcal{T}} Var_t$. We introduce by simultaneous inductive definition for every type $t$ the set $Expr_t$ of OCL expressions of type $t$. We will at the same time also define the set $free(e)$ of free variables for every expression $e$.

**Definition 7**

1. *If* $v \in Var_t$ *then* $v \in Expr_t$.
   $free(v) = \{v\}$.

2. *If* $s : \rightarrow t$ *is a constant symbol in* $\Omega$ *then* $s \in Expr_t$.
   $free(s) = \{\}$.

3. *If* $f \in \Omega$ *has signature* $f : t_1 \times \ldots \times t_n \rightarrow t$ *and* $e_i$, *for* $1 \leq i \leq n$ *are expressions of type* $t_i'$ *such that* $t_i' \preceq t_i$ *for all* $1 \leq i \leq n$ *then* $f(e_1, \ldots, e_n) \in Expr_t$.
   $free(f(e_1, \ldots, e_n)) = \bigcup_{1 \leq i \leq n} free(e_i)$

4. *If* $e_1 \in Expr_{Boolean}$ *and* $e_2$,$e_3$ *in* $Expr_{t_2}$, $Expr_{t_3}$ *such that* $t_2$ *and* $t_3$ *both conform to* $t$ *then* **if** $e_1$ **then** $e_2$ **else** $e_3$ **endif** *is in* $Expr_t$.
   $free($**if** $e_1$ **then** $e_2$ **else** $e_3$ **endif**$) = free(e_1) \cup free(e_2) \cup free(e_3)$.

5. *If* $v \in Var_{t_1}$, $e_1 \in Expr_{t_1}$, $e \in Expr_t$ *then*
   **let** $v = e_1$ **in** $e$ *is in* $Expr_t$.
   $free($**let** $v = e_1$ **in** $e) = free(e_1) \cup free(e) \setminus \{v\}$.

6. *If* $e_1 \in Expr_{Collection(t_1)}$, $v_1 \in Var_{t_1}$, $v_2 \in Var_t$, $e_2, e_3 \in Expr_t$ *then*
   $e_1 \rightarrow$ **iter**$(v_1; v_2 = e_2 \mid e_3)$ *is in* $Expr_t$.
   *This syntacial construct is subject to the variable conditions that* $v_i \notin free(e_1) \cup free(e_2)$ *for* $i = 1, 2$.
   $free(e_1 \rightarrow$ **iter**$(v_1; v_2 = e_2 \mid e_3)) = free(e_1) \cup free(e_2) \cup free(e_3) \setminus \{v_1, v_2\}$

7. *If* $e$ *is an expression and* $t$ *a type then* $(e$ **isTypeOf**$_\mathbf{t})$ $\in Expr_{Boolean}$ *and* $(e$ **isKindOf**$_\mathbf{t}) \in Expr_{Boolean}$ *with* $free(e$ **isTypeOf**$_\mathbf{t}) = free(e)$ *and* $free(e$ **isKindOf**$_\mathbf{t}) = free(e)$.

8. $(e$ **asType**$_\mathbf{t})$ `to be done`

## 5.3 Semantics of OCL Expressions

We will first describe the structures that will serve as semantic domains for the interpretation of OCL expressions and in the next subsection define the interpretation itself. At this point we only interpret OCL expressions

themselves, not the meaning of OCL expressions used as invariants or pre- and postconditions. This has to wait.

### 5.3.1 System States

A system state for the vocabulary $\Sigma = (\mathcal{T}, \Omega, <)$, usually denoted by $\sigma$ associates with every item $s$ from the vocabulary a semantic interpretation $\sigma(s)$. Here is the full definition

**Definition 8**

1. *For every type $t \in \mathcal{T}$ $\sigma(t)$ is a set.*
2. *For every function symbol $f : t_1 \rightarrow t_2$ in $\Omega$ the semantic interpretation $\sigma(s)$ is a function from $\sigma(t_1)$ to $\sigma(t_2)$.*

*The following constraints have to be satisfied by $\sigma$.*

1. *If $t_1 < t_2$ for $t_i \in \mathcal{T}$ then $\sigma(t_1) \subseteq \sigma(t_2)$.*
2. *$\sigma(OCLVoid) = \{\perp\}$ here $\perp$ serves as a special symbol for the undefined element.*
3. *For every $t \in \mathcal{T}$ its interpretation contains at least $\perp$, i.e. $\perp \in \sigma(t)$.*
4. *For every basic or enumeration type $t$ its interpretation $\sigma(t)$ has its fixed usual meaning plus $\perp$, e.g. $\sigma(Integer) = \mathbb{Z} \cup \{\perp\}$ etc.*
5. *The interpretation $\sigma$ respects the usual meaning of the collection type constructors. E.g. $\sigma(Set(t))$ is the set of all subsets of $\sigma(t)$, of course plus $\perp$.*
6. *$\sigma(OCLAny) = \bigcup\{\sigma(t) \mid t \in \mathcal{T}_B \cup \mathcal{T}_E \cup \mathcal{T}_C\}$*

### 5.3.2 System States Conforming to a Class Diagram

Let $\mathcal{D}$ be a class diagram, and $\Sigma^{\mathcal{D}} = (\mathcal{T}^{\mathcal{D}}, \Omega^{\mathcal{D}}, <^{\mathcal{D}})$ the vocabulary associated with it.

**Definition 9** *A system state $\sigma$ is said to* conform *to the class diagram $\mathcal{D}$ if is satisfies*

1. *The multiplicity constraints of all associations in $\mathcal{D}$.*
2. *If $C_1, \ldots, C_n$ are all direct subclasses of class $C$ in $\mathcal{D}$ and this subclassing is marked* disjoint *then the sets $\sigma(C_1), \ldots, \sigma(C_n)$ are mutually disjoint.*
3. *If $C_1, \ldots, C_n$ are all direct subclasses of class $C$ in $\mathcal{D}$ and this subclassing is marked* complete *then*

$$\sigma(C) = \sigma(C_1) \cup \ldots \cup \sigma(C_n).$$

4.

### 5.3.3 Interpreting OCL Expressions

In order to attach with an OCL expressen $e$ an unambigous meaning we need in addition to a fixed system state a mechanism to fix the meaning of free variables. Variable assigments will do this job.

**Definition 10**

1. *A variable assigment $\beta$ is a function defined on the set $Var$ of all variables such that for any $t \in \mathcal{T}$ and $x \in Var_t$ $\beta(x) \in \sigma(t)$.*
2. *An environment $I$ is a pair $(\sigma, \beta)$ of a system state $\sigma$ and a variable assigment $\beta$.*

We are now ready to explain rigorously the meaning of arbitrary OCL expressions with respect to a fixed environment $I$.

**Definition 11** *Let $I$ be a fixed environment $(\sigma, \beta)$. For any OCL expression $e$ of type $t$ its meaning $I(e)$ is an element of $\sigma(t)$ inductively defined as follows*

1. *If $e \in Var$ then $I(e) = \beta(e)$.*
2. *If $e$ is a constant symbol of type $t$ then $I(e) = \sigma(e)$.*
3. *If $e = f(e_1, \ldots, e_n)$ for some $f \in \Omega$ with signature $f : t_1 \times \ldots \times t_n \to t$ and $e_i \in Expr_{t_i'}$ such that $t_i' \preceq t_i$ for all $1 \leq i \leq n$ then $I(e) = \sigma(f)(I(e_1), \ldots, I(e_n))$ .*
4. *If $e = f(e_1, \ldots, e_n)$ for $f \in \Omega$ and $e_1, \ldots, e_n \in Expr$ then*
$$I(e) = \sigma(f)(I(e_1), \ldots, I(e_n))$$
5. *If $e =$ (**if** $e_1$ **then** $e_2$ **else** $e_3$ **endif**) for $e_1, e_2, e_3$ in $Expr$, then*
$$I(e) = \begin{cases} I(e_2) & \text{if } I(e_1 = true \\ I(e_3) & \text{if } I(e_1 = false \end{cases}$$
6. *If $e =$ (**let** $v = e_1$ **in** $e_0$) then*
$$I(e) = (\sigma, \beta_v^a)(e_0)$$
*with $a = I(e_1)$.*
*As usual $\beta_v^a(w) = \begin{cases} \beta(w) & \text{if } v \neq w \\ a & \text{if } v = w \end{cases}$*
7. *If $e = e_1 \to$ **iter**$(v_1; v_2 = e_2 \mid e_3)$ with $e_1 \in Expr_{Sequence(t_1)}$ and $I(e_1) = \langle a_1, \ldots, a_k \rangle$. Then*
$$I(e) = I_k(v_2)$$
*where $I_i = (\sigma, \beta_i)$ is defined as follows. For $w \notin \{v_1, v_2\}$ we have for all $0 \leq i \leq k$ $\beta_i(w) = \beta(w)$. The remaining values, $\beta_i(v_1)$ and $\beta_i(v_2)$, are defined by the following recursion:*
$$\begin{array}{lcll} \beta_0(v_1) & = & a_1 & \\ \beta_0(v_2) & = & I(e_2) & \\ \beta_{i+1}(v_1) & = & a_{i+1} & \text{for } 0 \leq i < k \\ \beta_{i+1}(v_2) & = & I_i(e_3) & \text{for } 0 \leq i < k \end{array}$$
*If $e_1$ has type Set or Bag then $I(e)$ is defined by choosing non-deterministically a sequence $s = \langle a_1, \ldots, a_k \rangle$ such that the set (resp. bag) of elements of $s$ equals $I(e_1)$.*

8. $I(e\ \mathbf{isTypeOf_t}) = true$ *if* $I(e) \in I(\mathbf{t})$ *and for all types* $\mathbf{t}'$ *with* $\mathbf{t}' < \mathbf{t}$ $I(e) \notin I(\mathbf{t}')$.

   *Otherwise* $I(e\ \mathbf{isTypeOf_t}) = false$.

   $I(e\ \mathbf{isKindOf_t}) = \begin{cases} true & \text{if } I(e) \in I(\mathbf{t}) \\ false & \text{otherwise} \end{cases}$

## 5.4 Comments

1. Discuss wether in Definition 7 Item 5 $v \notin free(e_1)$ should be added.

2. Item 3 in Definition 7

   - If $f \in \Omega$ has signature $f : t_1 \times \ldots \times t_n \to t$ and $e_i$, for $1 \leq i \leq n$ are expressions of type $t_i'$ such that for all $1 \leq i \leq n$
     (a) either $t_i' \preceq t_i$
     (b) or there is a type $t_i''$ with $t_i'' \preceq t_i$ and $t_i' = C(t_i'')$ where $C$ is one of the collection operators $Collection, Set, Bag, Sequence, Tupel$.

     then $f(e_1, \ldots, e_n) \in Expr_t$.
     $free(f(e_1, \ldots, e_n)) = \bigcup_{1 \leq i \leq n} free(e_i)$

   To get a picture of what is happening assume $g$ is a function mapping objects from the set $M$ to objects in $N$. Let furthermore $M_0 \subseteq M$. What is $g(M)$? First of all, we observe that there is a type mismatch, $g$ expects an argument of element type and is faced with a argument of set type. Nevertheless, it is common practise to accept $g(M)$ and attach as ist meaning the set $\{g(m) \mid m \in M\}$.

## 5.5 Exercises

# Chapter 6

# Metamodelling Approach to OCL

## 6.1 OCL Syntax Through Diagrams

Figure 6.1 shows the top level of the class hierarchy of OCL expressions. It is a very useful diagram to get an overview, what syntactical entities are involved, what are they named, how do they connect to the UML meta model. So we see, that every new syntactic entity introduced for OCL is a *model element* in the sense described in the core package of the UML meta model. Also we see, that OCL expressions come in 6 different froms. Detailed information will follow.

### 6.1.1 Comment

This information about OCLExp contained in Figure 6.1 could also be captured by a grammar rule

OclExpression ::= IfExp | VariableExp | LetExp |
PropertyCallExp | LiteralExp | OclMessageExp

where all occuring names are non-terminals.

## 6.2 IfExpression

The simplest of OCL expressions are If expressions. We use them to explain how to interpret diagrams as the one shown in Figure 6.2. This diagram explains, what is called the abstract syntax, in this case of If expressions. The diagram says, every If expression is made up of three component parts called *condition*, *thenExpression* and *elseExpression*. All parts are mandatory, occur at most once, and are OCL expressions. There however further constraints, not visible from the diagram in Figure 6.2. The condition part has to be an expression of type Boolean and the *thenExpression* and *elseExpression* must have comparable types, i.e. the type of one is a subtype of the other.

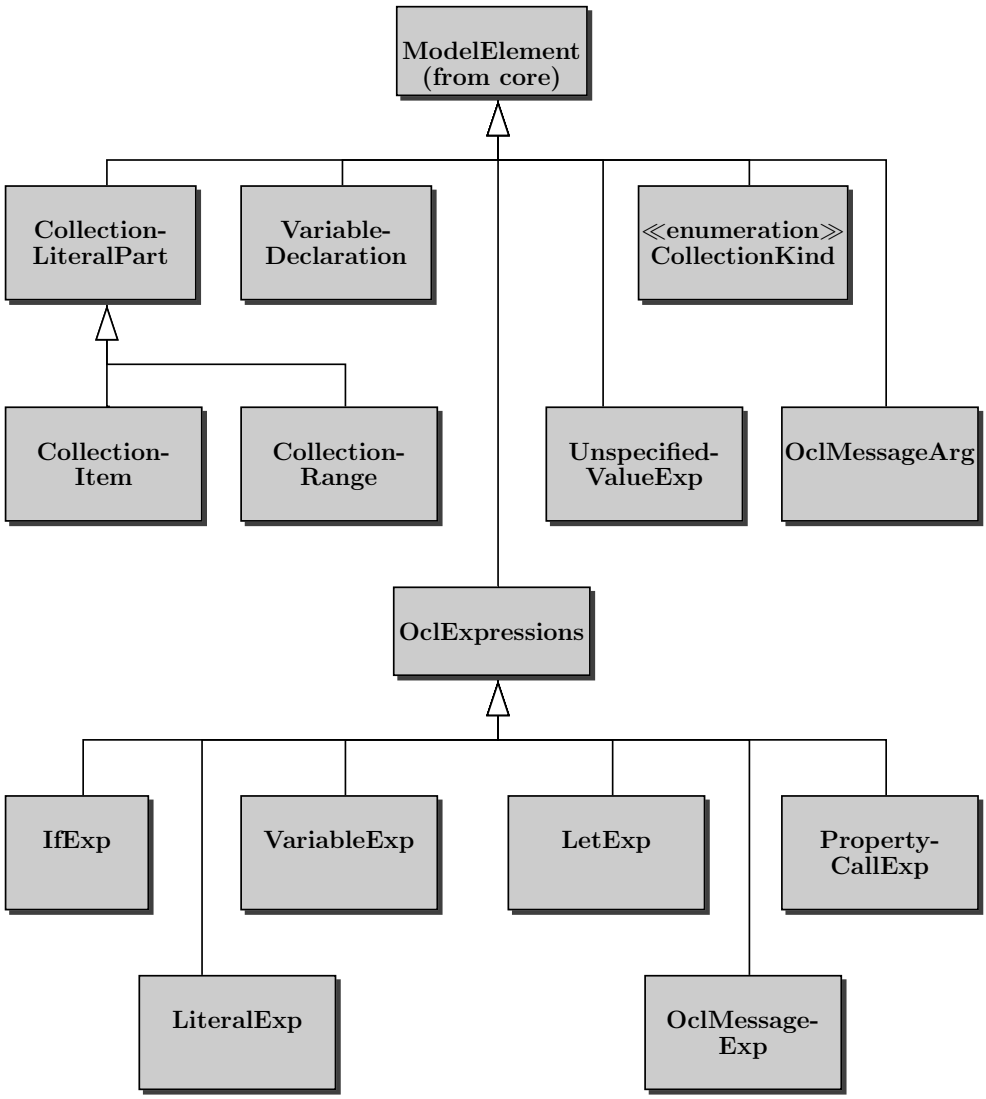



Figure 6.1: Top level meta model of OCL expressions

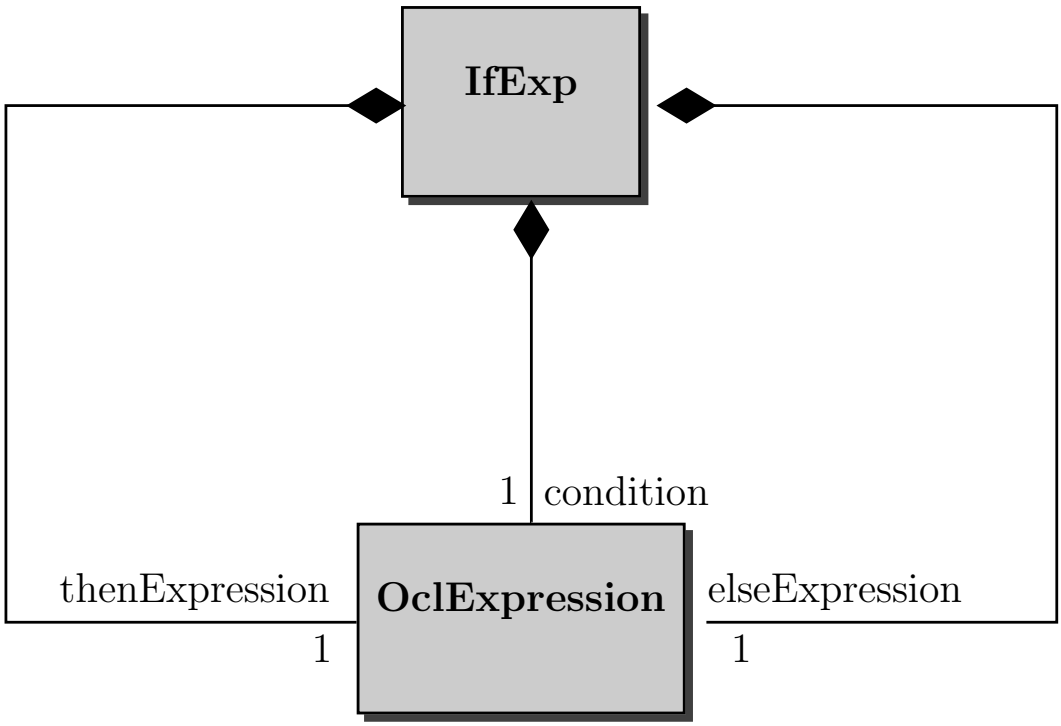



Figure 6.2: Class diagram for IfExpression

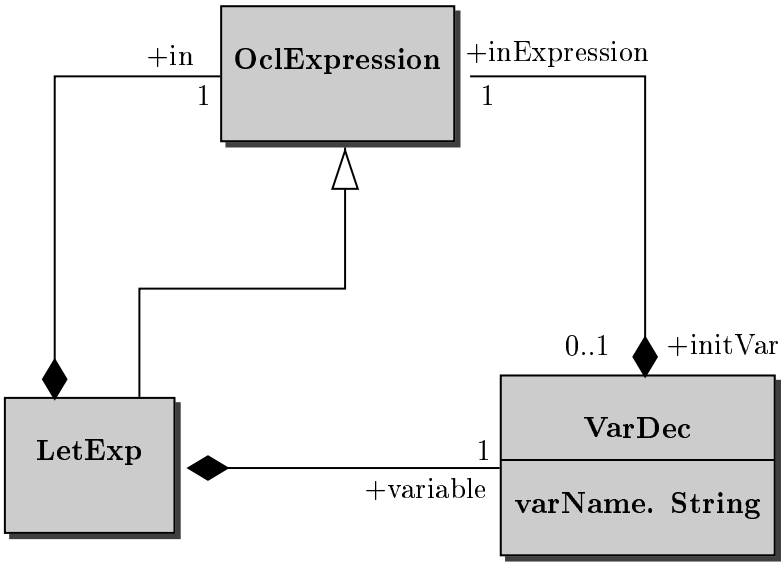



Figure 6.3: Class diagram for LetExpression

## 6.3 LetExpression

## 6.4 Exercises

# Chapter 7

# State Charts by Example

This Chapter contains an exposition of UML state charts.

## 7.1 States and Transitions

### 7.1.1 Example

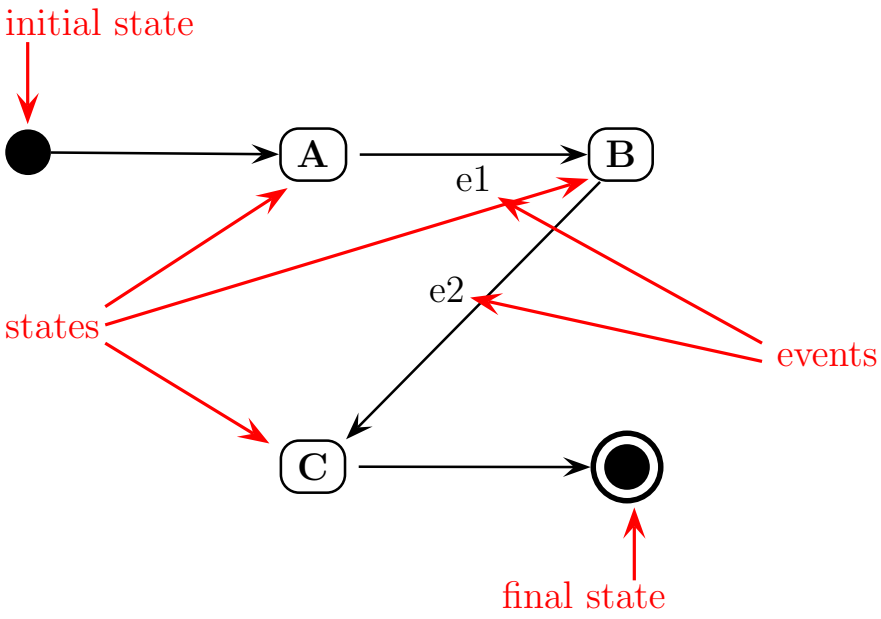



Figure 7.1: A simple State Chart

### 7.1.2 Description

Figure 7.1 shows the basic concepts of UML state charts: states, transitions, events. States are graphically represented as boxes, transitions as arrows and events occur as labels to transitions. There are two specials states in Figure 7.1: the initial state, denoted by a solid circle, and the final state, denoted by a solid circle surrounded by another circle. In UML state charts both

are special kinds of states, called *pseudo states*, and set apart from the other states. This is in contrast to most definitions of finite automata, where initial and final states are subsets of the set of all states. Transitions outgoing from pseudo states may not be labeled by events. There is no transition outgoing from the final state.

## 7.2 Completion States

### 7.2.1 Example

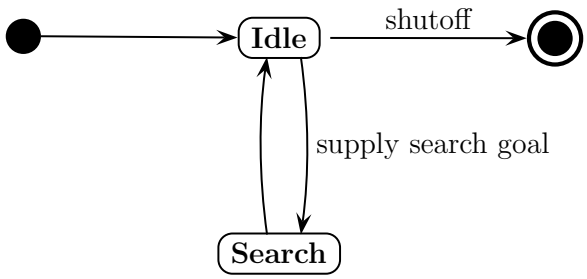



Figure 7.2: A State Chart with completioon state

### 7.2.2 Description

Figure 7.2 shows a state chart with a transition form state `search` to state `idle` without label. In this case `search` is called a completion state. The state itself is defined by the duration of an activity. In the case of Figure 7.2 we think of a search engine that is activated by the event of transmitting a search goal. The machine remains in state `search` as long as the search takes and then changes back to `idle`.

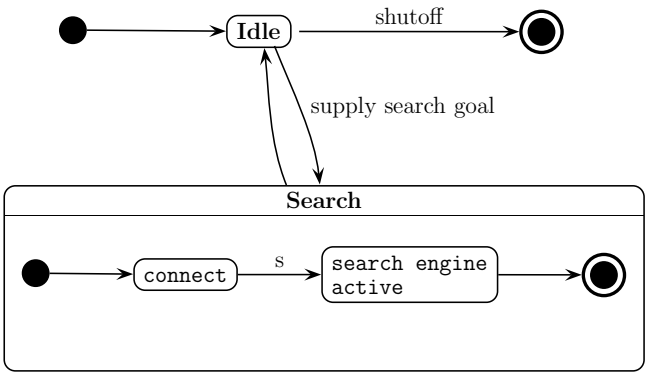



Figure 7.3: A State Chart with sequential substates

## 7.3 Sequential Substates

### 7.3.1 Example

### 7.3.2 Description

## 7.4 Concurrent Substates

### 7.4.1 Example

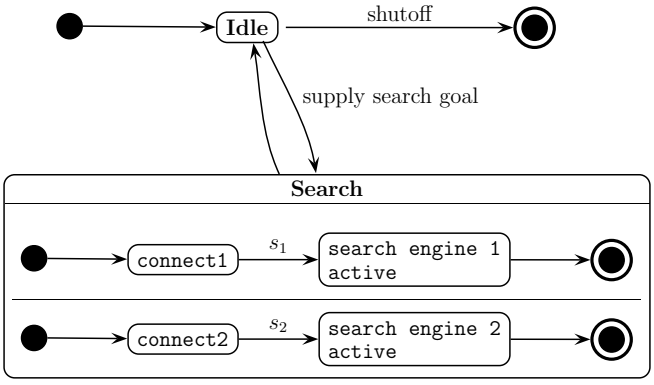



Figure 7.4: A State Chart with concurrent substates

### 7.4.2 Description

# Chapter 8

# Introduction to Abstract State Machines

## 8.1 A New Model of Sequential Computation

In this section we present an approach to a formal definition of computability, which extends the usual approaches using e.g. Turing Machines. This approach has been advocated by Yuri Gurevich since 1984. The earliest reference is probably [Gurevich, 1984]. The most concise exposition to date is [Gurevich, 2000]. We also used [Reisig, 2001].

### 8.1.1 The *Sequential Time* Postulate

**Definition 12 (Sequential Time Postulate)** *An algorithm $A$ is determined by*

1. *A set $\mathcal{S}(A)$ of states,*
2. *A subset $\mathcal{I}(A)$ of $\mathcal{S}(A)$, called the initial states of $A$,*
3. *A function $\tau_A : \mathcal{I}(A) \to \mathcal{I}(A)$, called the one-step transformation of $A$.*

**Definition 13 (Run)** *A run of an algorithm $A$ is a finite or infinite sequence $\langle X_i \rangle$ of states such that*

1. *$X_0$ is an initial state, i.e. $X_0 \in \mathcal{I}(A)$,*
2. *For every index $i$ in the sequence $\tau_A(X_i) = X_{i+1}$.*

**Comments** This definition incorporates a particular way of treating the end of a run, if the run terminates. A state $X \in \mathcal{S}(A)$ is final if $\tau_A(X)$ is not defined. One could alternatively have introduced an explicit set $\mathcal{F}(A)$ of final states. This is not a crucial decision.

### 8.1.2 The *Abstract State* Postulate

**Definition 14 (Abstract State Postulate)** *Let $A$ be an algorithm.*

1. *The states of $A$ are structures of first-order logic.*

2. *All states of $A$ have the same finite vocabulary, $\Sigma_A$.*

3. *The one-step transformation $\tau_A$ does not change the base set of any state.*

4. *The sets of states $\mathcal{S}(A)$ and $\mathcal{I}(A)$ are closed under isomorphism.*

5. *Any isomorphism between states $X$ and $Y$ is also an isomorphism between $\tau_A(X)$ and $\tau_A(Y)$.*

**Comments** The vocabulary $\Sigma_A$ will typically be many-sorted.

Requirement 4 is probably the least palatable. If we want to talk about an algorithm on the natural numbers $\mathbb{N}$, why should we be forced to consider isomorphic copies of $\mathbb{N}$? This also introduces the set theoretic problem, that the class of all isomorphic copies of $\mathbb{N}$ is not a set. Of course, there are standard ways to deal with this problem. But, could it not have been avoided alltogether? It turns out that all that is needed in the development of a theory of sequential algorithms so far is Lemma 3 below (see page 124). We suggest to take the claim of this Lemma as a postulate and waive closure under isomorphism.

### 8.1.3 The *Bounded Exploration* Postulate

**Definition 15 (Updates)** *Let $A$ be some algorithm, and $X$ a state in $\mathcal{S}(A)$.*

1. *A location of $X$ is a pair $(f, \bar{a})$ where $f$ is a function symbol and $\bar{a}$ is a sequence of elements from $X$. The length of the sequence $\bar{a}$ coincides with the arity of $f$.*

2. *If $(f, \bar{a})$ is a location for $X$ then $Content_X(f, \bar{a})$ is the element $f^X(\bar{a})$. Here $f^X$ is the interpretatin of the function symbol $f$ in $X$,*

3. *An update for $X$ is a tripel $(f, \bar{a}, b)$ where $(f, \bar{a})$ is a location for $X$ and $b$ is an element of $X$.*

4. *An update $(f, \bar{a}, b)$ is trivial if $b = Content_X(f, \bar{a})$.*

5. *Two updates $(f_1, \bar{a}_1, b_1)$, $(f_2, \bar{a}_2, b_2)$* clash *if $f_1 = f_2$, $\bar{a}_1 = \bar{a}_2$ and $b_1 \neq b_2$.*

6. *A set of updates $\Delta$ is consistent if if does not contain two clashing updates.*

**Definition 16** *Let $X$ be a state, $\Delta$ a consistent set of updates for $X$. Then $X + \Delta$ is the state $Y$ with the same base set as $X$ and the interpretation of function symbols $f$ given by:*

$$f^Y(\bar{a}) = \begin{cases} f^X(\bar{a}) & \text{if there is no } c \text{ such that } (f, \bar{a}, c) \in \Delta, \\ b & \text{if } (f, \bar{a}, b) \in \Delta. \end{cases}$$

**Lemma 2** *If $X, Y$ are two first-order structures with the same vocabulary and the same base set then there is a unique consistent set $\Delta$ of non-trivial updates such that $Y = X + \Delta$*

**Proof** Obvious, see also [Gurevich, 2000].

∎

**Definition 17** *If $A$ is an algorithm given by $(\mathcal{S}(A), \mathcal{I}(A), \tau_A)$, then we denote for every $X \in \mathcal{S}(A)$ the unique set $\Delta$ satisfying $X + \Delta = \tau_A(X)$ by $\Delta(A, X)$.*

**Definition 18 (Bounded Exploration Postulate)** *Let $A$ be an algorithm. Then there exists a finite set $T$ of (ground) terms in the vocabulary of $\Sigma_A$ such that for any two states $X$, $Y$ such that*

$$t^X = t^Y \text{ for all } t \in T$$

*then*

$$\Delta(A, X) = \Delta(A, Y)$$

*The set $T$ is called the set of critical terms for $A$. In [Gurevich, 2000] $T$ is called a* bounded exploration witness *for $A$.*

The following definition sums up our considerations so far

**Definition 19 (Sequential Algorithm)** *A sequential algorithm $A$ is an object determined by $(\mathcal{S}(A), \mathcal{I}(A), \tau_A)$ satisfying the postulates 12, 14 and 18.*

**Comments**

**Lemma 3** *Let $A$ be an algorithm and $T$ its finite set of critical terms and $T_1$ the smallest set of terms containing $T$ and all subterms of terms in $T$.*

*For any state $X \in \mathcal{S}(A)$ and any update $(f, (a_1, \ldots, a_j), a_0) \in \Delta(A, X)$ there is for every $i$, $0 \leq i \leq j$ a term $t_i \in T_1$ such that $t_i^X = a_i$.*

**Proof** Assume for the sake of a contradiction that for some $i$, $0 \leq i \leq j$ we have $a_i \neq t^X$ for all $t \in T_1$. Let $Y$ be an isomorphic copy of $X$ arising from replacing $a_i$ by a new element $b$, i.e. $b$ does not occur in the base set of $X$ and $a_i$ does not occur in the base set of $Y$. By the postulate 14 $Y$ is also a state in $\mathcal{S}(A)$. It is easy to see that for all $t \in T_1$ we have $t^X = t^Y$. Thus by 18 we have $\Delta(A, X) = \Delta(A, Y)$. Thus $(f, (a_1, \ldots, a_j), a_0) \in \Delta(A, Y)$, but this contradicts the fact the $a_i$ is not in the base set of $Y$.

This is basically the argument given in [Gurevich, 2000, Lemma 6.2].

∎

**Corollary 4** *Let $A$ be an algorithm. Then there is a finite number $n$ such that for all states $X$ the set $\Delta(A, X)$ has not more than $n$ elements.*

**Proof** Let $T$ be the finite set of critical terms of $A$ and $T_1$ the smallest set of terms containing $T$ and all subterms of terms in $T$. Obviously $T_1$ is still finite. From this the finiteness claim obviously follows.

∎

### 8.1.4 Example: A Geometric Algorithm

One of the advantages of the definition of an algorithm given in Definition 19 is the fact that no coding in natural numbers or any other fixed data structure is required. An algorithm may be described at any chosen level of abstraction. We want to illustrate this by a geometric algorithm $A_{3C}$, which determines for three given points $P_1$, $P_2$, $P_3$ in the plane the centre $M$ of the circle that hits all three points.

The structures in $\mathcal{S}(A_{3C})$ will be many-sorted structure with the sorts $Point$, $Line$, $Circle$, and $PairOfPoints$. The intuitive meaning of the last sort,

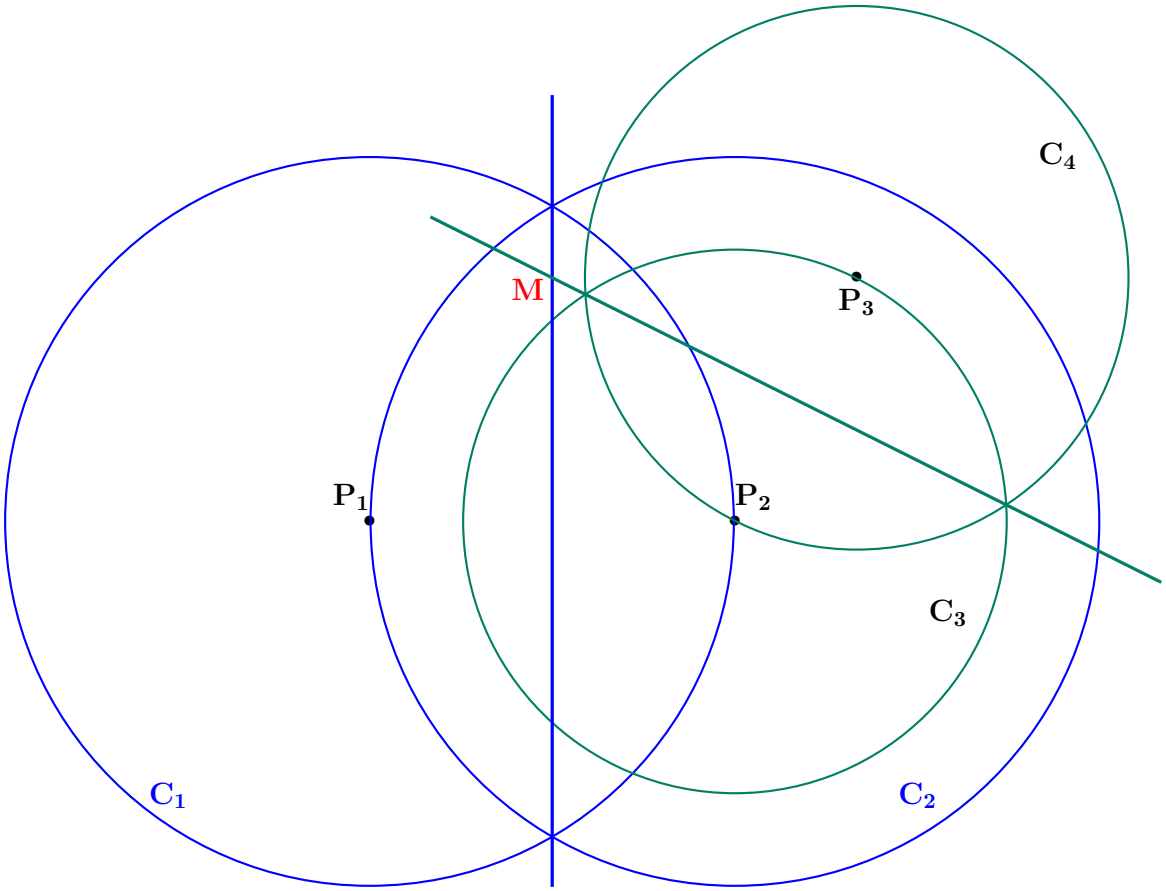



Figure 8.1: Constructing the centre point $M$

$PairOfPoints$, is a set of at most two points. Thus also the case of a one-element set of points and also the empty set of points is included.

Next, we have to decide which geometric operations we assume to be available to determine the point $M$? Roughly speaking, we will allow all operations that can be effected with compass and ruler. More precisely the vocabulary $\Sigma_{A_{3C}}$ consists of

$$
\begin{array}{ll}
circle & : Point \times Point \times Point \to Circle \\
intersectL & : Line \times Line \to Point \\
line & : PairOfPoints \to Line \\
intersectC & : Circle \times Circle \to PairOfPoints \\
P_1 & : \to Point \\
P_2 & : \to Point \\
P_3 & : \to Point \\
M & : \to Point
\end{array}
$$

For any structure $\mathcal{U} \in \mathcal{S}(A_{3C}) = (U, I)$ we define that

1. $circle(Q_1, Q_2, Q_3)$ is the circle with centre $Q_1$ and radius $r$ given by the distance between $Q_2$ and $Q_3$. If $Q_2 = Q_3$ the circle is identified with the singleton set $\{Q_1\}$.

2. $intersectL(L_1, L_2)$ is the point, in which the lines $L_1$, $L_2$ intersect or $undef$ if $L_1$, $L_2$ are parallel.

3. $line(S)$ is the uniquely defined line through the points $Q_1$, $Q_2$ if $S = \{Q_1, Q_2\}$ with $Q_1 \neq Q_2$, and $undef$ otherwise.

4. $intersectC(C_1, C_2)$ is the set of points common to $C_1$ and $C_2$. If $C_1 = C_2$ then $intersectC(C_1, C_2)$ is $undef$.

A structure $\mathcal{U} \in \mathcal{S}(A_{3C}) = (U, I)$ is an initial structure of $A_{3C}$, i.e. $\mathcal{U} \in \mathcal{I}(A_{3C})$ if

1. If $I(P_1)$, $I(P_2)$, $I(P_3)$ are points not lying on a line.

2. $I(M)$ is undefined.

For structures $\mathcal{U} \in \mathcal{I}(A_{3C})$ the one-step transformation $\tau_{3C}(\mathcal{U}) = \mathcal{W}$, where the only change in $\mathcal{W}$ with respect to $\mathcal{U}$ is in the interpretation of the constant

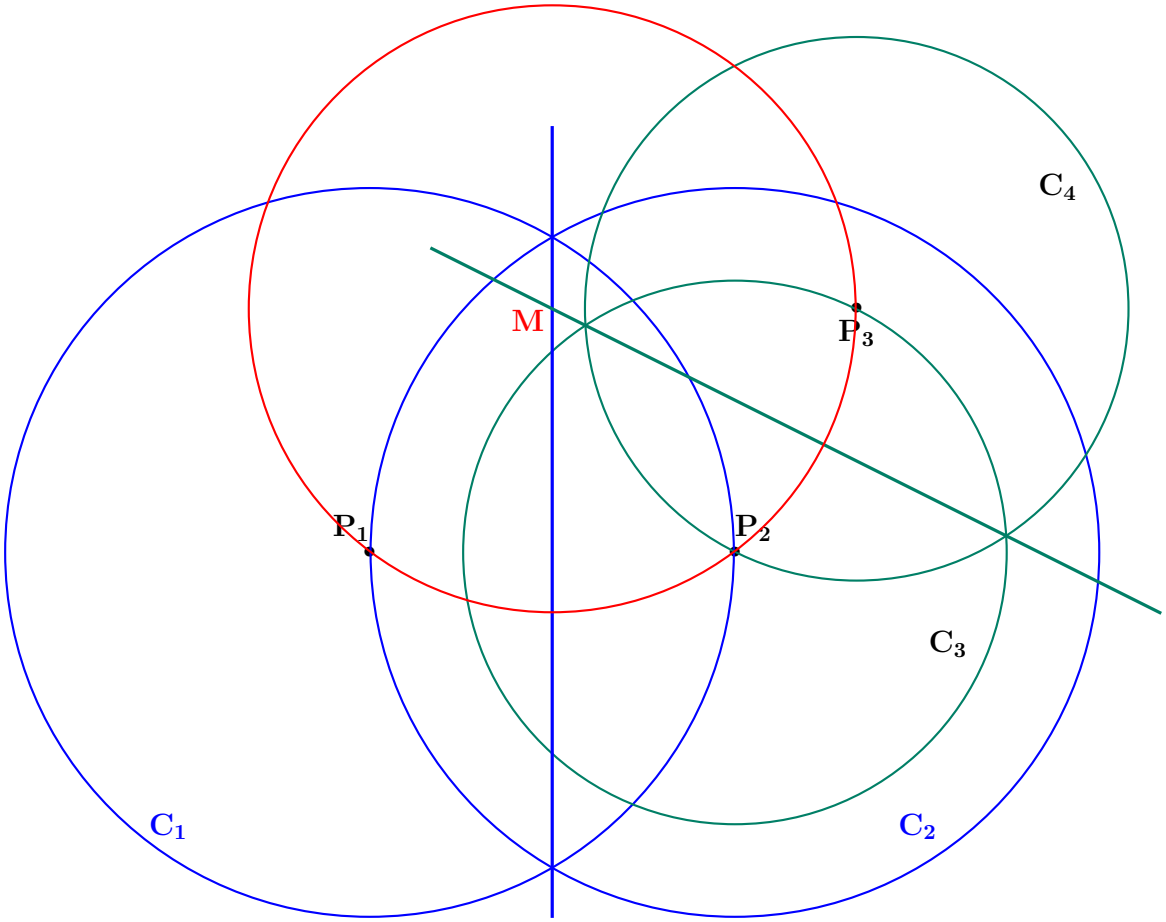



Figure 8.2: The circle touching three given points

$M$. This is given by:

$$I^W(M) = intersectL^U(l_1, l_2)$$

with

$$\begin{aligned} l_1 &= line^U(intersectC^U(C_1, C_2)) \\ l_2 &= line^U(intersectC^U(C_3, C_4)) \\ C_1 &= circle^U(P_1, P_1, P_2) \\ C_2 &= circle^U(P_2, P_1, P_2) \\ C_3 &= circle^U(P_2, P_2, P_3) \\ C_4 &= circle^U(P_3, P_2, P_3) \end{aligned}$$

$\tau_{3C}$ is not defined for non-initial structures. Thus runs for the algorithm $A_{3C}$ are very simple: they start with an initial structure $\mathcal{U}_0 \in \mathcal{I}(A_{3C})$, proceed with $\mathcal{U}_1 = \tau_{3C}(\mathcal{U}_0)$, and that is the end. Some simple reasoning shows that $I^{U_1}(M)$ is defined. Thus $\tau_{3C}(\mathcal{U}_1)$ is not defined, the run is completed.

### 8.1.5 What Is A Single Step?

One of the constituent features of Definition 19 is the one-step transformation $\tau$. But what exactly is *one step*? Most steps can be broken down into simpler steps. Is there a notion of atomic steps? We want to shed some light on this issue by the following example.

The vocabulary $\Sigma_{sq}$ of algorithm $A_{sq}$ contains only one sort $Int$ and a constant symbol *const*. The typical structure in $\mathcal{S}(A_{sq})$ has as it universe $U$ the set of integers There is no restriction on $I^{\mathcal{U}}(const)$. All of $\mathcal{S}(A_{sq})$ is obtained by taking isomorphic copies. We further stipulate $\mathcal{I}(A_{sq}) = \mathcal{S}(A_{sq})$. For $\mathcal{U} \in \mathcal{S}(A_{sq})$ the one-step transformation $\tau_{sq}(\mathcal{U}) = \mathcal{W}$ is obtained by

$$I^{\mathcal{W}}(const) = I^{\mathcal{U}}(const)^2$$

This makes sense immediately for the typical structures with universe equal to the set of integers. In the remaining cases the effect of the squaring operation is mimicked by isomorphic transfer. Is $A_{sq}$ given by $(\mathcal{S}(A_{sq}), \mathcal{I}(A_{sq}), \tau_{sq})$ a sequential algorithm? The structure $\mathcal{U}$ with universe the set of integers and $I^{\mathcal{U}}(const) = 2$. is certainly in $\mathcal{S}(A_{sq})$ and $\Delta(A_{sq}, \mathcal{U}) = \{(const, \langle\rangle, 4)\}$. The set of critical terms $T$ can at most be $T = \{const\}$. This would contradict Lemma 3. Thus $A_{sq}$ is not a sequential algorithm.

Let the vocabulary $\Sigma^+_{sq}$ be $\Sigma_{sq} \cup \{+\}$. In structures $\mathcal{U} \in \mathcal{S}A^+_{sq}$ the symbol $+$ is interpreted as addition. Otherwise the algorithm $A^+_{sq}$ coincides with $A_{sq}$. By the same argument $A^+_{sq}$ still does not satisfy the requirements of a sequential algorithm. Only when we extend the vocabulary further to include also multiplication $*$ will $A^{+,*}_{sq}$ satisfy the requirement of Definition 19.

This seems to contradict our experience, that we can write an algorithm computing the square function by using addition only. The crucial point to observe here, is the fact that the notion of one-step transformation does exlude loops. Here is the correct set-up of an algorithm $A_{square}$ computing the square function by using addition only.

The vocabulary $\Sigma_{square}$ contains the two sorts $Int$ and $Bool$ and the function symbols

$$
\begin{array}{ll}
input & : \rightarrow Int \\
output & : \rightarrow Int \\
counter & : \rightarrow Int \\
0 & : \rightarrow Int \\
+ & : Int \times Int \rightarrow Int \\
= & : Int \times Int \rightarrow Bool \\
< & : Int \times Int \rightarrow Bool \\
- & : Int \rightarrow Int \\
minus1 & : Int \rightarrow Int \\
\& & : Bool \times Bool \rightarrow Bool
\end{array}
$$

The universe of all structures $\mathcal{U}$ in $\mathcal{S}(A_{square})$ is the disjoint union of the set of all integers and the Boolean values $\{\mathbf{true}, \mathbf{false}\}$. The symbols 0, $+$, $=$, $<$, $-$, & are interpreted as usual and also $minus1$ will be as expected, i.e. $I^{\mathcal{U}}(minus1)(n) = n - 1$. $I(input)$ may be any number. For initial structures $\mathcal{U}$ we require $I^{\mathcal{U}}(output) = I^{\mathcal{U}}(counter) = 0$. For $\mathcal{U} \in \mathcal{S}(A_{square})$ the one-step transformation function $\tau_{square}(\mathcal{U})$ equals $\mathcal{W}$. The only symbols whose interrpetation may possibly change in passing from $\mathcal{U}$ to $\mathcal{W}$ are $input$, $counter$ and $output$.

$$
\begin{array}{lll}
\text{if } input < 0 & \text{then} & I^{\mathcal{W}}(input) = I^{\mathcal{U}}(-(input)) \\
\text{if } input > 0 \ \& \ counter = 0 & \text{then} & I^{\mathcal{W}}(counter) = I^{\mathcal{U}}(input) \\
\text{if } input > 0 \ \& \ counter > 0 & \text{then} & I^{\mathcal{W}}(counter) = I^{\mathcal{U}}(minus1(counter)) \\
 & & I^{\mathcal{W}}(output) = I^{\mathcal{U}}(output + input)
\end{array}
$$

It can be easily checked that for every initial structure $\mathcal{U} \in \mathcal{I}(A_{square})$ there is only one run. This run is finite and for its final state $\mathcal{W}$ we get $I^{\mathcal{W}}(output)$

is the square of $I^{\mathcal{U}}(input)$.

### 8.1.6 Example: A Graph Algorithm

We set out to describe an algorithm $A_{G0}$ for testing reachability within graphs. On our way to an eventual solution we will discover some subtleties of definition 19.

The states of $A_{G0}$ will be two-sorted first-order structures, containting the sorts $Bool$ and $Node$. The universe $U$ of any structure in $\mathcal{S}(A_{G0})$ will be the disjoint union $U = U_{Bool} \cup U_{Node}$. Where $U_{Bool}$ always equals $\{\mathbf{true}, \mathbf{false}\}$, while $U_{Node}$ may be any non-empty set.

The vocabulary $\Sigma_{G0}$ contains the following function symbols

$$
\begin{array}{ll}
edge & : Node \times Node \rightarrow Bool \\
start & : \rightarrow Node \\
finish & : \rightarrow Node \\
true & : \rightarrow Bool \\
false & : \rightarrow Bool \\
reachable & : Node \rightarrow Bool
\end{array}
$$

$\mathcal{S}(A_{G0})$ consists of all $\Sigma_{G0}$-structures $\mathcal{U} = (U, I)$ such that $U_{Bool} = \{\mathbf{true}, \mathbf{false}\}$, $I(true) = \mathbf{true}$, $I(false) = \mathbf{false}$, and $U_{Node} \neq \emptyset$.

$\mathcal{I}(A_{G0})$ consists of all $\mathcal{U} \in \mathcal{S}(A_{G0})$ with

1. $\{n \in U_{Node} \mid I(reachable)(n) = \mathbf{true}\} = \{I(start)\}$, i.e. in an inital state the start node is the only node marked *reachable*.

Finally for $\mathcal{U} = (U, I_U) \in \mathcal{S}(A_{G0})$ the one-step transformation $\tau_{G0}$ of $A_{G0}$ has to be defined. If $I_U(reachable(finish)) = \mathbf{true}$, then $\tau_{G0}(\mathcal{U})$ is not defined. Otherwise $\tau_{G0}(\mathcal{U}) = \mathcal{W} = (W, I_W)$ with

for all $n \in U_{Node}$ $I_W(reachable)(n) = \mathbf{true}$ iff $I_U(reachable)(n) = \mathbf{true}$ or there is $n_0 \in U_{Node}$ with $I_u(edge)(n_0, n) = \mathbf{true}$ and $I_U(reachable)(n_0) = \mathbf{true}$.

Is $A_{G0}$ thus defined an algorithm? If yes, then there exists by the postulate 18 a set of critical terms $T$. The terms in $T$ are of no particular interest to us at this moment, we are only interested in the number $k$ of elements in $T$.

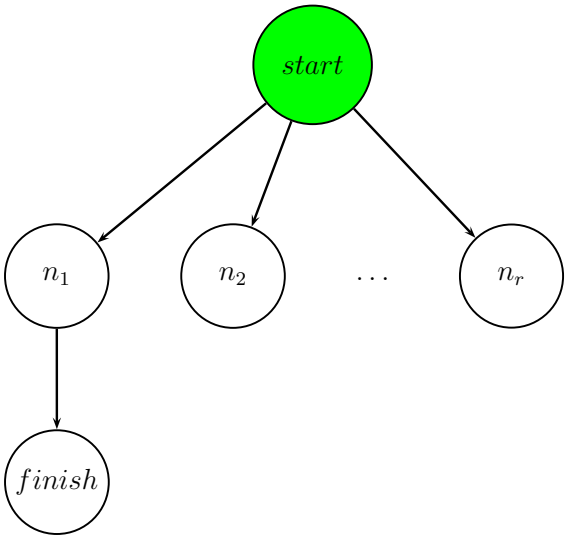



Figure 8.3: Example of a reachability problem

Look at the graph in Figure 8.3 where we have chosen $r$ (the number of successor nodes of start) to be greater than $k$. Let $\mathcal{U}_0$ be the structure in $\mathcal{I}(A_{G0})$ whose graph part is given by the graph in Figure 8.3 and $\mathcal{U}_1 = \tau_{G0}(\mathcal{U}_0)$. Then $\Delta(A_{G0}, \mathcal{U}_0)$ contains at least the locations

$$\{(reachable, \langle n_1 \rangle, \mathbf{true}), \ldots, (reachable, \langle n_r \rangle, \mathbf{true})\}$$

By Lemma 3 there exists a term $t_i \in T$ such that $t_i^{\mathcal{U}_0} = n_i$ for all $1 \leq i \leq r$. Since all the nodes $n_i$ are different, $r > k$ and there are only $k$ elements is $T$, this is a contradiction. $A_{G0}$ is not a sequential algorithm?

The problem here is that $A_{G0}$ does too many updates in parallel.

`muss ab hier noch ergänzt werden`

## 8.2 ASM Programs

In the previous section we have introduced a precise and abstract notion of algorithm (see Definition 19). Part of this definition was the one-step

transformation $\tau$. There was no restriction on how $\tau$ should be defined, as long as this was done unambigously. In this section we will turn to an approach that is closer to main stream computer science. We will describe an abstract, programing language. Programs in this language go by the name of *rule* or more precisely *Abstract State Machine rule* (ASM rule, for short).

We then consider the same set-up as in Definition 12, with the only, but notable difference, that the transformation if effected by a program or ASM rule.

### 8.2.1 Definition

**Definition 20 (ASM Rule)** *Let $\Sigma$ be a fixed vocabulary. We assume that Bool is a sort of $\Sigma$. A term in $\Sigma$ is called a* Boolean term *if it is of sort Bool.*

1. *If $\phi$ is a Boolean term and $R_1$, $R_2$ are rules then*

$$\textbf{if } \phi \textbf{ then } R_1 \textbf{ else } R_2 \textbf{ endif}$$

   *is a rule,*

2. *If $R_1, \ldots R_k$ are rules then*

$$\begin{array}{l}\textbf{par}\\ R_1\\ \vdots\\ R_k\\ \textbf{endpar}\end{array}$$

   *is a rule,*

3. *If $f$ is an $n$-place function symbol in $\Sigma$, $t_0, t_1, \ldots t_n$ are $\Sigma$-terms then*

$$f(t_1, \ldots t_n) := t_0$$

   *is a rule.*

For any ASM rule $R$ and any $\Sigma$-structure $\mathcal{U}$ in $\mathcal{S}$ the result of applying $R$ in the state $\mathcal{U}$, denoted by $\tau_R(\mathcal{U})$, will be defined as $\mathcal{U} + R(\mathcal{U})$ for a set of updates $R(\mathcal{U})$.

**Definition 21 (Update of Rules)** *Let $R$ be an ASM rule of vocabulary $\Sigma$ and $\mathcal{U}$ a $\Sigma$-structure.*

1. *If $R$ is of the form $f(t_1, \ldots t_n) := t_0$ then*
$$R(\mathcal{U}) = \{(f, \langle a_1, \ldots, a_n \rangle, a_0)\}$$
*where $a_i = t_i^{\mathcal{U}}$. Here $(f, \bar{a}, a_0)$ is a location, see Definition 15.*

2. *If $R$ is of the form*
**par**
$R_1$
$\vdots$
$R_k$
**endpar**
*then*
$$R(\mathcal{U}) = R_1(\mathcal{U}) \cup \ldots \cup R_k(\mathcal{U})$$

3. *If $R$ is of the form* **if** $\phi$ **then** $R_1$ **else** $R_2$ **endif** *then*
$$R(\mathcal{U}) = \begin{cases} R_1(\mathcal{U}) & \text{if } \phi^{\mathcal{U}} = \textbf{true} \\ R_2(\mathcal{U}) & \text{otherwise} \end{cases}$$

**Definition 22 (Transformation of an ASM Rule)** *Let $R$ be an ASM rule and $\mathcal{U}$ an eligible structure then*
$$\tau_R(\mathcal{U}) = \mathcal{U} + R(\mathcal{U})$$
*For an explanation of of "+", see Definition 16. If $R\mathcal{U})$ is inconsistent then $\tau_R(\mathcal{U})$ is not defined.*

**Definition 23 (Sequential Abstract State Machine)** *A* sequential Abstract State Machine *$B$ of vocabulary $\Sigma$ is given by:*

1. *A ASM program $R$ of vocabulary $\Sigma$,*
2. *A set $\mathcal{S}(B)$ of $\Sigma$-structures closed under isomporphism,*
3. *A subset $\mathcal{I}(B)$ of $\mathcal{S}(B)$, called the initial states of $B$,*

**Lemma 5** *Let $B$ be a sequential Abstract State Machine of vocabulary $\Sigma$ given by $(\mathcal{S}(B), \mathcal{I}(B), R)$ then $(\mathcal{S}(B), \mathcal{I}(B), \tau_R)$ is an sequential algorithm (see Definition 19).*

**Proof:** Easily checked.

■

### 8.2.2 Examples

As a first example let us look again at the one-step transformation $\tau_{square}$ from Subsection 8.1.5.

**par**
  **if** $(input < 0)$ **then** $input := -(input)$
  **endif**
  **if** $(input > 0 \ \& \ counter = 0)$ **then** $counter := input$
    **else par**
      counter := minus1(counter)
      output := output + input
      **endpar**
  **endif**
**endpar**

As a convention let us agree that rules, or updates that are written vertically one below the other are implicitely understood to be executed in parallel. Then the above ASM rule looks:

**if** $(input < 0)$ **then** $input := -(input)$ **endif**
**if** $(input > 0 \ \& \ counter = 0)$ **then** $counter := input$
  **else** $counter := minus1(counter)$
    $output := output + input$
**endif**

### 8.2.3 Universality of Abstract State Machines

The purpose of this subsection is to prove:

**Theorem 6** *Let $A$ be a sequential algorithm of vocabulary $\Sigma_A$ given by $(\mathcal{S}(A), \mathcal{I}(A), \tau_A)$ then there is a ASM rule $R$ of vocabulary $\Sigma_A$ such that $\tau_R$ coincides with $\tau_A$ on $\mathcal{S}(A)$, i.e.*

$$(\mathcal{S}(A), \mathcal{I}(A), \tau_A) = (\mathcal{S}(A), \mathcal{I}(A), \tau_R^*)$$

*if $\tau_R^*$ denotes the restriction of $\tau_R$ to $\mathcal{S}(A)$.*

This theorem has first been formulated and proved in [Gurevich, 2000].

**Lemma 7** *For any state $X$ of $A$ there is a ASM rule $R^X$ such that $R^X = \Delta(X)$.*

**Proof:** This is easy. Assume $\Delta(X) = \{(f_i, \langle a_1^i, \ldots, a_{n_i}^i \rangle, b^i) \mid i = 1 \ldots k\}$ By the accessability lemma 3 there are terms $s_j^i$ and $t^i$ such that $(s_j^i)^X = a_j^i$ and $(t^i)^X = b^i$. Thus $R^X$:

**par**
$f_1(s_1^1, \ldots, s_{n_1}^1) := t^1$
$\vdots$
$\vdots$
$f_k(s_1^k, \ldots, s_{n_k}^k) := t^k$
**endpar**

satisfies $R^X(X) = \Delta(X)$

■

Note, that $R^X(Y) = \Delta(Y)$ does not hold for arbitrary $Y$.

**Definition 24 ($T$-similar states)** *Let $T$ be a set of ground terms. Two states $X$,$Y$ are called $T$-similar if for ayn pair $t_1, t_2 \in T$*

$$t_1^X = t_2^X \Leftrightarrow t_1^Y = t_2^Y$$

**Lemma 8** *If $X$, $Y$ are $T_A$-similar states of the algorithm $A$, then*

$$R^X(Y) = \Delta(Y)$$

**Proof** First assume that $X$ and $Y$ are disjoint. Let $Z$ be the structure isomorphic to $Y$, where the universe of $Z$ is obtained by replacing every element $t^Y$ by $t^X$ for all $t \in T$. By the assumption of the lemma the function $F : Y \to Z$:

$$F(y) = \begin{cases} y & \text{if } y \neq t^Y \text{ for all } t \in T \\ t^X & \text{if } y = t^Y \end{cases}$$

is surjective and injective and may thus be extended to an isomorphism from $Y$ onto $Z$, as intended. Since for all $t \in T$ we have $t^X = t^Z$ we get by the defining property of the set of critical terms $\Delta(Z) = \Delta(X) = R^X(X)$. From this we see immediately $\Delta(Y) = R^X(Y)$.

∎

# Chapter 9

# Introduction to Dynamic Logic

For a thorough introduction to Dynamic Logic we recommend [Harel, 1984] or [Kozen & Tiuryn, 1990] or the recent book [Harel *et al.*, 2000] . A nice example how to use propositional dynamic logic may be found in [Shilov & Yi, 2001].

We start our introduction in the next section by an example that shows how Dynamic Logic can be used to verify programs. Treatment of the example will appeal more to intution than precise definitions. This will be made up for in the remaining sections of this chapter where the individual parts of Dynamic logic will looked at separately and rigorous definitions will be provided.

## 9.1 A Motivating Example

```
int a, b, z;
z = 0;
while (b! = 0)
        {  if ((b/2) * 2 == b)
                {a = 2 * a;
                b = b/2; }
        else
                {z = z + a;
                a = 2 * a;
                b = b/2; }
        }
```

Figure 9.1: The Program $\alpha_{RM}$

Figure 9.1 shows a program written in a simple *while* language. Variable declarations, type information and input-output functions have been omitted or kept to a minimum at this level of abstraction. An executable Java version of this program is given in Appendix 16.1. The algorithm implemented by this program is sometimes refered to as Russian peasant's multiplication. We abbreviate the program in Figure 9.1 by $\alpha_{RM}$ and will use it to illustrate the basic concepts of Dynamic Logic and to demonstrate program verification.

To express formal properties of programs assertions are used. These are frequently placed within the program code, see Figure 9.2. An assertion is understood to be a statement containing a boolean expression that the programmer believes to be true at the time the statement is executed.

```
int a, b, z;
z = 0;
assert x ≐ a ∧ y ≐ b;
while (b! = 0)
assert a ∗ b + z ≐ x ∗ y;
        {  if ((b/2) ∗ 2 == b)
                {a = 2 ∗ a;
                b = b/2; }
        else
                {z = z + a;
                a = 2 ∗ a;
                b = b/2; }
        }
assert b ≐ 0;
assert z ≐ x * y;
```

Figure 9.2: The Program $\alpha_{RM}$ with assertions

It is a very typical situation that one wants to assert that after some computation the input values have been processed in some specific manner. E.g. in Figure 9.2 we assert that after the while-loop the program variable $z$ equals the value of the product of the variables $a$ and $b$ before the while-loop was entered. This can be accomplished by the use of new variables, sometimes called external variables. This are $x$ and $y$ in the above example. At the beginning of the program the external variables are declared equal to the program variables. Furthermore external variables are chosen such that they do not occur within the program itself, i.e. they will be changed by the program.

Dynamic Logic takes a different approach. Rather than using formulas within programs it uses programs within formulas. This proves more flexible in many cases and allows to express properties that cannot be obtained by program annotation.

Instead of placing assert $F$; after some program segment $\alpha$, we write $[\alpha]F$. Later, also the form $\langle\alpha\rangle F$ will be introduced and the differences explained.

If programm $\alpha_{RM}$ is started in a program state where the values of the program variables $a$ and $b$ are the integers $A$ and $B$, then it is supposed to terminate with the value of $z$ equal to the product $A * B$ of $A$ and $B$. This claim written as a Dynamic Logic formula reads as follows:

$$\forall a^{int}, b^{int}, x^{int}, y^{int}, z^{int}(x \doteq^{int} a \wedge y \doteq^{int} b \rightarrow [\alpha_{RM}]\ z \doteq^{int} x * y) \tag{9.1}$$

How would we prove such a statement? It is a sensible strategy to unravel the program $\alpha_{RM}$. At a first step we get $\alpha_{RM} = z = 0; \alpha_{RMwhile}$. Here $\alpha_{RMwhile}$ denotes the rest of the program $\alpha_{RM}$ after the first assigment command has been chopped off. In this section we apply proof rules intuitively. Later we will see formal definitions of these rules. Obviously, after the command $z = 0$ has been executed the value of $z$ at the beginning of the execution of $\alpha_{RMwhile}$ equals 0. This the claim 9.1 reduces to

$$\begin{aligned}\forall a^{int}, b^{int}, x^{int}, y^{int}, z^{int} \quad & (x \doteq^{int} a \wedge y \doteq^{int} b \wedge z \doteq^{int} 0 \\ & \rightarrow [\alpha_{RMwhile}]\ z \doteq^{int} x * y)\end{aligned} \tag{9.2}$$

The next job is to unravel the while-program $\alpha_{RMwhile}$. One way to do this is via invariants. The general pattern goes as follows: to prove that after execution of a while program while$(B)\{body\}$ the statement $A$ is true one searches for another statement $Inv$, called an *invariant* such that

1. $Inv$ is true before execution of the while-program.
2. $Inv \wedge B \rightarrow [body]Inv$ can be proved.
3. $Inv \wedge \neg B \rightarrow A$ is true.

For the program $\alpha_{RMwhile}$ the invariant

$$a * b + z \doteq^{int} x * y$$

works. Thus claim 9.2 will be established once the following three claims have been proved:

$$\begin{aligned}\forall a^{int}, b^{int}, x^{int}, y^{int}, z^{int} \quad & (x \doteq^{int} a \wedge y \doteq^{int} b \wedge z \doteq^{int} 0 \\ & \rightarrow a * b + z = x * y)\end{aligned} \tag{9.3}$$

$$\forall a^{int}, b^{int}, x^{int}, y^{int}, z^{int} \quad (a * b + z = x * y \rightarrow [\alpha_{RMbody}]\ a * b + z = x * y) \tag{9.4}$$

$$\forall a^{int}, b^{int}, x^{int}, y^{int}, z^{int} \quad (a * b + z = x * y \wedge b \doteq 0 \rightarrow z = x * y) \tag{9.5}$$

The claims 9.3 and 9.4 are obvious, so we will concentrate on 9.4 for the rest. Here $\alpha_{RMbody}$ is the body of the while loop, i.e.

$$\text{if}((b/2) * 2 == b)\{a = 2 * a; b = b/2;\}\,\text{else}\{z = z + a; a = 2 * a; b = b/2;\}$$

Notice, that 9.4 deviates from the general pattern in that the condition ($b! = 0$) of the while loop does not occur at the lefthand side of the implication. The reason is, that it will not be needed in the argument. So we omitted it from the start.

Notice that the branching condition of the if statement $(b/2) * 2 == b$ just is a round-about way to express that $b$ is even. Let us abbreviate the program $\alpha_{RMbody}$ as

$$\text{if}(even(b))\{branch_0\}\,\text{else}\{branch_1\}$$

A formula of the form $[\alpha_{RMbody}]A$ is then treated as a case distinction and decomposed into two claims, one for each branch of the if construct.

$$\forall a^{int}, b^{int}, x^{int}, y^{int}, z^{int} \quad (a * b + z = x * y \wedge even(b) \rightarrow [\alpha_{branch_0}]\ a * b + z = x * y) \tag{9.6}$$

$$\forall a^{int}, b^{int}, x^{int}, y^{int}, z^{int} \quad (a * b + z = x * y \wedge odd(b) \rightarrow [\alpha_{branch_1}]\ a * b + z = x * y) \tag{9.7}$$

The remaining programs $\alpha_{branch_0}$ and $\alpha_{branch_1}$ only contain assigment commands. We can mimic the effect of these programs by performing the substitutions corresponing to these assigments in the formula following the box operator. This does not work in general, but it suffices here. This leads us to the following to claims:

$$\forall a^{int}, b^{int}, x^{int}, y^{int}, z^{int} \quad (a * b + z = x * y \wedge even(b) \rightarrow (2 * a) * (b/2) + z = x * y) \tag{9.8}$$

$$\forall a^{int}, b^{int}, x^{int}, y^{int}, z^{int} \quad (a * b + z = x * y \wedge odd(b) \rightarrow (2 * a) * (b/2) + z + a = x * y) \tag{9.9}$$

### 9.1.1 Comments

The assert construct supported by Java 1.4 is not a full-blown design-by-contract facility, it can help support an informal design-by-contract style of programming. So, one should be careful carrying the similarities of this programming language construct with Dynamic Logic formulas too far.

The method of invariants used above is but one method to handle while-loops. Sometimes it is very tricky to find an invariant that does the job. Other methods, e.g. symbolic evaluation, will be considered later.

## 9.2 Prerequisites

The chapters and sections upto this one were written to be accessible without a background in logic. This is no longer possible for the aims of the present chapter. We will assume that the reader is familiar with the basic concepts of first-order predicate logic. In particular he should understand the items in the following list

- a signature $\Sigma$ specifies a set $Func$ of functions symbols and a set $Pred$ of predicate symbols. Function symbols of arity 0 are called constant symbols. Predicate symbols of arity 0 are treated like propositional variables.
- By $Term$, or more precisely $Term_{Sigma}$ we denote the set of all terms in signature $\Sigma$.
- By $Fml$, or more precisely $Fml_{\Sigma}$ we denote the set of all first-order formulass in signature $\Sigma$.
- If $x$ is a variable, $t, s$ are terms in $Term$, $A$ a formula in $Fml$ then the term $s(t/x)$ results from $s$ by replacing every occurrence of $x$ in $s$ by $t$. $F(t/x)$ is obtained from $F$ by replacing every free occurrence $x$ in $F$ by $t$.
- Structures for first-order predicate logic are denote by $\mathcal{M}$ or variants and consist of a universe $M$ and an interpretation function $I$, $\mathcal{M} = (M, I)$.

- Variable assignments are usually denoted by $\beta$,
- Given a state $(\mathcal{M} = (M, I), \beta)$, the interpretation $(I, \beta)(t)$ of a term $t \in \mathcal{T}$ is an element from $M$.
- Given a state $(\mathcal{M} = (M, I), \beta)$ it is possible to evaluate the truth value $(I, \beta)(\phi)$ of any formula $\phi \in \mathcal{FML}$. $(\mathcal{M}, \beta) \models \phi$ is equivalent to $(I, \beta)(\phi) = 1$.
- We distinguish bound and free variables.
- It is not strictly necessary but would help to know the basic syntax and semantics of modal logic.

## 9.3 The Vocabulary

In Section 9.1 we have already seen examples of formulas of Dynamic Logic. So far we depend on a rather vague understanding of the meaning of these formulas. A precise definition of the semantics of Dynamic Logic still has to wait till section 9.6. Here we are only concerned with the syntax. Two observations are important here. There are syntactic constructs, the angular $\langle\rangle$ and square brackets [ ], that go beyond first-order logic. These are modal operators. The text within brackets of both kinds is obviously very different from the text outside. The first category is called the program part, the second the logical part.

In modal logic the modal operators occur in the simple forms $\Box$, $\Diamond$, or in multi-modal logics in the indexed forms $\Box_i$, $\Diamond_i$. In Dynamic Logic modal operators are much more complex. For every program $\pi$ there will be modal operators $\langle\pi\rangle$ and $[\pi]$.

### 9.3.1 Parts of the Vocabulary

We will present right from the start a typed version of Dynamic Logic. To fix a particular instance of Dynamic Logic we have to decide on a *vocabulary*. The vocabulary comes in two parts

- A vocabulary for the logical part,

- A vocabulary for constructing programs,

The vocabulary for the logical part consists of

1. A set Type of types,
2. A set $\Sigma$ of function and relation symbols.

$\Sigma$ itself is made up of different parts

$$\Sigma = \Sigma^f_{nr} \cup \Sigma^f_r \cup \Sigma^r_{nr} \cup \Sigma^r_r$$

$\Sigma^f_{nr}$ contains the non-rigid function symbols, $\Sigma^f_r$ the rigid function symbols and respectively $\Sigma^r_{nr}$ and $\Sigma^r_r$ the non-rigid and rigid relation symbols. Intuitively rigid function and relation symbols do not change in the system model under consideration. Typically, one may think of functions and relations on data types, like addition and the $\leq$-relation on integers. With every $n$-ary function symbol $f$ we associate the types $s_1, \ldots, s_n$ of its arguments and its value type $s$. This information is frequently presented in the form $f : s_1 \times \ldots \times s_n \to s$. With every $n$-ary relation symbol $r$ we associate the types of its arguments, $r : s_1 \times \ldots \times s_n$.

Equality $\doteq^s$ is a predicate symbol, always assumed to be present in $\Sigma^r_r$ for every type $s \in$ Type.

For every type $s \in$ Type there will be an unlimited reservoir of variables $x^s, y^s, x^s_i, y^s_k$ of type $s$.

Function symbols may have 0 arguments, then they are called constants. Also relation symbols may have 0 arguments, then they are called propositional variables.

The syntax of the program part offers a much wider range of possibilities. Possible choices are the language of regular programs (see examples 3 and 8 in Exercise 9.18.2), the while programs one encounters in computation theory (see 1 and 5 in Figure 9.18.2) or real world programming languages , like Java ( 2 in Figure 9.18.2). Little can be said at this general level.

We should however not fail to mention one important distinction, that between propositional and first-order Dynamic Logic. In propositional Dynamic Logic there is analogous to the concept of a propositional variable on the logical side the concept of an atomic program on the program side. A

propositional variable has no internal structure, all we know is that it may have the value **true** or **false** . Likewise, atomic programs have no internal structure, all they do is to establish a connection between initial and final state in some previously fixed state space.

The set of atomic programs, $\Pi_0$, is the only item of the program part of the vocabulary, that we will assume to be always present at this stage of our presentation.

### 9.3.2 Example

By $\mathcal{L}_r$ we denote the instance of Dynamic Logic needed to talk about the example program in Figure 9.1 . The vocabulary of $\mathcal{L}_r$, as defined above, is given as follows. It is to be implicitely understood that all parts ot the vocabulary considered in this subsection belong to $\mathcal{L}_r$. This saves us putting an extra index $_r$ on every symbol.

The set Type of types for $\mathcal{L}_r$ is

$$\text{Type} = \{int, bool\}$$

There are only rigid function and relation symbols, i.e.

$$\Sigma = \Sigma_r = \Sigma_r^f \cup \Sigma_r^r.$$

$$\begin{array}{lllllllllll}
\Sigma_r^f: & != & : & int \times int & \to & bool & == & : & int \times int & \to & bool \\
 & / & : & int \times int & \to & int & 0 & : & & \to & int \\
 & * & : & int \times int & \to & int & 2 & : & & \to & int \\
 & + & : & int \times int & \to & int & & & & &
\end{array}$$

$\Sigma_r^r$ contains only the two equality relations $\doteq^{int}$ and $\doteq^{bool}$.

### 9.3.3 Comments

The *vocabulary* of an instance $\mathcal{L}$ of Dynamic Logic is also called the *signature* of a $\mathcal{L}$.

For a vocabulary $\Sigma = \Sigma_{nr}^f \cup \Sigma_r^f \cup \Sigma_{nr}^r \cup \Sigma_r^r$ we will also use the abbreviations $\Sigma_r = \Sigma_r^f \cup \Sigma_r^r$ and $\Sigma_{nr} = \Sigma_{nr}^f \cup \Sigma_r^r$. A systematic treatment of non-rigid

functions in Dynamic Logic, despite the fact that it offers little extra difficulties, has not been accomplished yet. On the other hand, special cases of non-rigid functions, in the form of e.g. arrays, have been covered extensively.

In this section we are only concerned with what is called the *abstract* syntax, i.e. we only talk about what categories of syntactical objects exist. *Concrete* syntax, on the other hand, determines details such as: function names should start with a lowercase or uppercase letter, a function may or may not have the same name as a relation, a unary function may or may not have the same name as a binary function etc.

## 9.4 Formulas and Terms of Dynamic Logic

### 9.4.1 Definitions

We will define terms and formulas for the instance $\mathcal{L}_r$ of Dynamic Logic with logical vocabulary $\Sigma$ and Type as set of types and **while**-programs as the set of allowed programs. This is fairly general. The only restriction is in the choice of the program part.

**Definition 25** *For every type $s \in \text{Type}$ the set $\text{Term}^s_{\mathcal{L}_r}$ of all terms of type $s$ is inductively defined by:*

1. *Every variable $x^s$ is in $\text{Term}^s_{\mathcal{L}_r}$.*
2. *If $f : s_1 \times \ldots \times s_n \to s$ is a function symbol in $\Sigma$ and $t_i$ are terms in $\text{Term}^{s_i}_{\mathcal{L}_r}$ then $f(t_1, \ldots, t_n)$ is a term in $\text{Term}^s_{\mathcal{L}_r}$.*

The set $\text{Term}^s_{\mathcal{L}_r}$ does not depend on the allowed programs.

A term $t$ is called *flexible* or *non-rigid* if it starts with a non-rigid function symbol, i.e. $t = f(t_1, \ldots, t_n)$ with $f \in \Sigma_{nr}$. The set of formulas $\text{Fml}_{\mathcal{L}_r}$ and programs $\Pi_{\mathcal{L}_r}$ are defined by simultaneous induction:

**Definition 26**

1. *If $r : s_1 \times \ldots \times s_n$ is a relation symbol in $\Sigma$ and $t_i$ are terms in $\text{Term}^{s_i}_{\mathcal{L}_r}$ then $r(t_1, \ldots, t_n)$ is a formula in $\text{Fml}_{\mathcal{L}_r}$.*

2. *If* $t_1, t_2$ *are terms of type* $s$ *then* $t_1 =^s t_2$ *is in* $\mathrm{Fml}_{\mathcal{L}_r}$ .

3. *If* $F_1, F_2$ *are in* $\mathrm{Fml}_{\mathcal{L}_r}$ *then also* $F_1 \vee F_2$ $F_1 \wedge F_2$, $F_1 \rightarrow F_2$, $\neg F_1$, $\forall x^s F_1$ *and* $\exists x^s F_1$ *are in* $\mathrm{Fml}_{\mathcal{L}_r}$.

4. *If* $F$ *is a formula in* $\mathrm{Fml}_{\mathcal{L}_r}$ *and* $\pi$ *is a program in* $\Pi_{\mathcal{L}_r}$ *then* $[\pi]F$ *and* $\langle\pi\rangle F$ *are formulas in* $\mathrm{Fml}_{\mathcal{L}_r}$ .

5. *Every atomic program* $\alpha$ *in* $\Pi_0$ *is a program in* $\Pi_{\mathcal{L}_r}$.

6. *If* $t$ *is a term of type* $s$ *and* $x$ *is a variable of type* $s$ *then* $x = t$ *is in* $\Pi_r$.

7. *If* $t$ *is a term of type* $s$ *and* $f$ *is a flexible term of type* $s$ *then* $f = t$ *is in* $\Pi_r$.

8. *If* $\pi_1$,$\pi_2$ *are in* $\Pi_r$ *then also* $\pi_1; \pi_2$ *is in* $\Pi_r$.

9. *If con is in* $\mathrm{Term}^{bool}_{\mathcal{L}_r}$ *and* $\pi$ *a program in* $\Pi_r$ *then*

$$\text{while } (con)\ \{\pi\}$$

*is a program in* $\Pi_r$.

10. *If con is in* $\mathrm{Term}^{bool}_{\mathcal{L}_r}$ *and* $\pi_1, \pi_2$ *are programs in* $\Pi_r$ *then*

$$\text{if } (con)\ \{\pi_1\}\ \text{ else } \{\pi_2\}$$

*is in* $\Pi_r$.

### 9.4.2 Examples

The following are well-formed formulas of $\mathcal{L}_r$:

$\forall a^{int} \forall b^{int} \forall z^{int}\ \langle \alpha_{RM} \rangle$ **true**
$\forall x^{int} \forall y^{int} \forall a^{int} \forall b^{int} \forall z^{int} (\ x \doteq^{int} a \wedge y \doteq^{int} b \rightarrow [\alpha_{RM}]\ z \doteq^{int} x * y)$
$\forall x^{int} \forall y^{int} \forall a^{int} \forall b^{int} \forall z^{int} (\ x \doteq^{int} a \wedge y \doteq^{int} b \rightarrow \langle \alpha_{RM} \rangle\ z \doteq^{int} x * y)$

### 9.4.3 Comments

Item 7 in Definition 26 is not part of the usual definition of Dynamic Logic. The program in Figure 9.1 contains no example of a non-rigid (i.e. flexible) symbols. The first example of these will occur in Section 9.10.

## 9.5 Kripke Structures for Dynamic Logic

### 9.5.1 Definitions

A Kripke structure in general consists of states, sometimes also called worlds, and an accessibility relation between states. In the case of multi-modal logics there is one accessibility relation for every modality. In the case of Kripke structures for Dynamic Logic, DL-Kripke structures for short, there will be an accessibility relation, usually denoted by $\rho(\pi)$, for every program $\pi$. The state space of the Kripke structures for an instance of Dynamic Logic depends first of all on the allowed programs and on the vocabulary. For proposional Dynamic Logic a state is nothing more than a truth value assignment to the propositional variables. If only imperative programs are considered and all functions and relations are rigid states may be identified with assignments to the program variables. Below we give the definition of Kripke structures for $\mathcal{L}_r$. In this case states are identified with structures of typed first-order predicate logic.

**Definition 27** *A DL-Kripke structure $\mathcal{K} = (S, \rho)$ consists of a set $S$ of states (or worlds) and a function $\rho$ that maps every atomic program $\pi$ to a binary relation $\rho(\pi)$ on $S$. The $\rho(\pi)$ are called the accessibility relations.*

*For the Dynamic Logic $\mathcal{L}_r$ with vocabulary $\Sigma = \Sigma_r \cup \Sigma_{nr}$ the set $S$ of $\mathcal{K}$ is subject to the following requirements*

- *There is a $\Sigma_r$-structure $\mathcal{A}_r$ called the rigid part of $\mathcal{K}$,*
- *$S$ consists of pairs $(\mathcal{A}, \beta)$, where $\mathcal{A}$ is a $\Sigma$-structure and $\beta$ a variable assignement, i.e. a function from the set of alle variables $Var$ into the universe of $\mathcal{A}_r$,*
- *For every pair $(\mathcal{A}, \beta)$ in $S$ the restriction of $\mathcal{A}$ to the signature $\Sigma_r$ equals $\mathcal{A}_r$, in symbols $\mathcal{A}_{|\Sigma_r} = \mathcal{A}_r$.*

### 9.5.2 Examples

For any Kripke structure $\mathcal{A}$ for $\mathcal{L}_r$ the rigid part $\mathcal{A}_r$ coincides with the whole structure, i.e. $\mathcal{A} = \mathcal{A}_r$, because $\Sigma = \Sigma_r$. The universe $A_r$ of $\mathcal{A}_r$ is the

disjoint union of $\mathbb{Z}$ and $\mathbb{B}$, i.e. $A_r = \mathbb{Z} \cup \mathbb{B}$ and $\mathbb{Z} \cap \mathbb{B} = \emptyset$, with $\mathbb{Z}$ the set of integers and $\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$. The interpretation of the functions and relation symbols from $\Sigma$ is standard. Only integer division needs, maybe, an explanation:

$$a/b = \text{ the greatest integer } x \text{ with } x * b \leq a.$$

Thus $8/2 = 4$ and $7/2 = 3$.

For any Kripke structure $\mathcal{K} = (S, \rho)$ of $\mathcal{L}_r$ the set of states $S$ consists of all pairs $(\mathcal{A}, \beta)$, where $\mathcal{A}$ is the fixed structure described above and $\beta$ is, as usual, a variable assignment. If we ignore atomic programs for the moment there is only one possible function $\rho$ interpreting the program constructs used to build $\Pi_r$ in the usual way. Here are some examples. Since the first component of all states is fixed we identify $S$ with the set of all variable assignments.

$$\rho(z^{int} = a^{int}) = \{(\beta, \beta') \mid \beta'(z) = \beta(a)\}$$

### 9.5.3 Comments

Notational variations of the definition of Kripke structures occur in the literature. Most notably, Kripke structures are occasionally presented as tripels $\mathcal{K} = (S^0, I, \rho)$, where $I$ is an interpretation function that associates structures $I(s)$ with states $s \in S^0$. In this case $S^0$ is just an abstract set. In Definition 27 we have set, loosely speaking, $S = \{I(s) \mid s \in S^0\}$. This is always possible if $\rho$ does depend on $I(s)$ only, i.e. if $(I(s_1), I(s_2) \in \rho(\pi)$ and $I(s_i) = I(t_i)$ implies $(I(t_1), I(t_2) \in \rho(\pi)$.

Definition 27 contains a principle decision on the basic definition of a Kripke structure. It endorses the concept of a *constant domain* Kripke structure, i.e. all structures occuring as the first component of a pair in $S$ share a common universe, the universe $A$ of $\mathcal{A}_r$. The theory of Kripke structures with variable domains is decisively more complicated than the constant domain case and not much is know about it. (For the best account to date, see the book [Fitting & Mendelsohn, 1999]). What is the price to be paid for working with constant domains? Do we not sometimes want to add new objects to a class? and does this not contradict the constant domain principle? The solution, that has also been adopted in other similar situations is as follows: The constant domain universe of the structures as part of states is

seen as an infinite reservoir of possible elements. Adding new elements is achieved via a $0-1$-function $ex$. $ex(obj) = 1$ means that object $obj$ exists, $ex(obj) = 0$ means that $obj$ sits in the reservoir waiting to be born. A new object is created by picking an element $c$ from the reservoir with $ex(c) = 0$ and changing the value of $ex$ to $ex(c) = 1$. Usually the situation will be more specific. Let $customer$ be a $0-1$-function. Adding a new customer now amounts to picking $c$ from the reservoir with $customer(c) = 0$ and apply the function update $customer(c) = 1$.

Our definition of allowed programs $\Pi_r$ in $\mathcal{L}_r$, see clauses 5 to 10 in Definition 26, is rather unusual in that it mixes atomic programs, that play a crucial role in Propositional Dynamic Logic, with programs based on assignment statements, that make essential use of the presence of program and logical variables. This will prove convenient for our purposes in translating UML into $\mathcal{L}_r$. We will come back to this topic in greater detail, but we can give here a preview. In UML class diagrams operations may be introduced. At this modeling level an operation $op$ is just a reference to an otherwise unspecified program. Later $op$ may be implemented by a , e.g. Java method. The semantics of this situation is best captured by translating $op$ into an atomic program in Dynamic Logic.

## 9.6 Truth Definition in Kripke Structures

### 9.6.1 Definitions

For the next definition it helps to remember that a structure $\mathcal{A}$ for first-order predicate logic is of the form $\mathcal{A} = (A, I)$, where $A$ is the universe of $\mathcal{A}$ and the function $I$ associates with every item in the vocabulary its semantic interpretation in $\mathcal{A}$. In particular, for a term $t$ with variables $x_1, \ldots, x_k$ its interpretation $I(t)$ is an $n$-ary function from $A$ into $A$. For every type symbol $s \in \text{Type}$ $I(s)$ is a subset of $A$. For two types $s_1$, $s_2$ either $I(s_1) \subseteq I(s_2)$ holds, when $s_1$ is declared a subtype of $s_2$, or $I(s_2) \subseteq I(s_1)$ holds, when $s_2$ is declared a subtype of $s_1$, or $I(s_1) \cap I(s_1) = \emptyset$ otherwise. If necessary we write $I_{\mathcal{A}}$ instead of $I$.

**Definition 28** *Let $\mathcal{K} = (S, \rho)$ be a given DL-Kripke structure, $(\mathcal{A}, \beta)$ a state in $S$, $F$ a formula in $\text{Fml}_{\mathcal{L}_r}$ and $\pi$ a program in $\Pi_r$. The evaluation $t^{(\mathcal{A},\beta)}$*

*of terms $t$ is defined as usual. We will define by simultaneous induction the semantics of $\pi$, $\rho(\pi)$, and $(\mathcal{A},\beta) \models F$, $F$ is true in state $(\mathcal{A},\beta)$ of $\mathcal{K}$:*

1. $(\mathcal{A},\beta) \models r(t_1,\ldots,t_k)$ *iff* $(t_1^{(\mathcal{A},\beta)},\ldots,t_k^{(\mathcal{A},\beta)}) \in I(r)$.
   *(Remember, $\mathcal{A} = (A, I)$.)*
2. $(\mathcal{A},\beta) \models t_1 = t_2$ *iff* $t_1^{(\mathcal{A},\beta)} = t_2^{(\mathcal{A},\beta)}$
3. $(\mathcal{A},\beta) \models F$ *is defined as usual if the principal logical operator of $F$ is one of the classical operators $\wedge$, $\vee$, $\rightarrow$, $\neg$, or one of the quantifiers $\forall$, $\exists$.*
4. $(\mathcal{A},\beta) \models \langle p\rangle F$ *iff there is a pair* $((\mathcal{A},\beta),(\mathcal{B},\gamma)) \in \rho(p)$ *with* $(\mathcal{B},\gamma) \models F$.
5. $(\mathcal{A},\beta) \models [p]F$ *iff* $(\mathcal{B},\gamma) \models F$ *for all pairs* $((\mathcal{A},\beta),(\mathcal{B},\gamma))$ *of states in* $\rho(p)$.
6. *If $x$ is a variable,* $\rho(x := s) = \{((\mathcal{A},\beta),(\mathcal{A},\beta[x/s^{(\mathcal{A},\beta)}])) \mid (\mathcal{A},\beta) \in S\}$.
7. *If* $t = f(t_1,\ldots,t_n)$ *is a non-rigid term, then* $\rho(t := s)$ *consists of all pairs* $((\mathcal{A},\beta),(\mathcal{B},\beta))$ *such that $\mathcal{B}$ coincides with $\mathcal{A}$ except for the interpretation of $f$, which is given by*

$$f^{\mathcal{B}}(b_1,\ldots,b_n) = \begin{cases} s^{(\mathcal{A},\beta)} & \text{if } (b_1,\ldots,b_n) = (t_1^{(\mathcal{A},\beta)},\ldots,t_n^{(\mathcal{A},\beta)}) \\ f^{\mathcal{A}}(b_1,\ldots,b_n) & \text{otherwise} \end{cases}$$

8. $\rho(\pi_1;\pi_2)$ *consists of all pairs* $((\mathcal{A},\beta),(\mathcal{C},\gamma))$ *such that* $((\mathcal{A},\beta),(\mathcal{B},\delta)) \in \rho(\pi_1)$ *and* $((\mathcal{B},\delta),(\mathcal{C},\gamma)) \in \rho(\pi_2)$.
9. $((\mathcal{A},\beta),(\mathcal{B},\gamma)) \in \rho(\text{while}(F_0)\{\pi\})$ *iff there is an* $n \in \mathbb{N}$ *and there are states* $(\mathcal{A}_i,\beta_i)$ *for* $0 \leq i \leq n$ *such that*
   (a) $(\mathcal{A}_0,\beta_0) = (\mathcal{A},\beta)$,
   (b) $(\mathcal{A}_n,\beta_n) = (\mathcal{B},\gamma)$,
   (c) $(\mathcal{A}_i,\beta_i) \models F_0$ *for* $0 \leq i < n$
   (d) $(\mathcal{A}_n,\beta_n) \models \neg F_0$
   (e) $((\mathcal{A}_i,\beta_i),(\mathcal{A}_{i+1,\beta}\, i+1)) \in \rho(\pi)$ *for* $0 \leq i < n$
10. $((\mathcal{A},\beta),(\mathcal{B},\gamma)) \in \rho(\text{if}(F_0)\{\pi_1\}\,\text{else}\{\pi_2\})$ *iff* $(\mathcal{A},\beta) \models F_0$ *and* $((\mathcal{A},\beta),(\mathcal{B},\gamma)) \in \rho(\pi_1)$ *or* $(\mathcal{A},\beta) \models \neg F_0$ *and* $((\mathcal{A},\beta),(\mathcal{B},\gamma)) \in \rho(\pi_2)$

### 9.6.2 Examples

### 9.6.3 Comments

Definition 28 is as usual, see e.g. [Kozen & Tiuryn, 1990], with the exception of clause 7.

## 9.7 Some DL Tautologies

**Definition 29 (DL Tautology)** *A Dynamic Logic formula $F$ is called a tautology if it is true in every state of every DL-Kripke structure.*

**Definition 30 (Deterministic Programs)**
*A program $\pi$ is called* deterministic *iff for its interpretation $\rho(\pi)$ holds that $(s, s_1) \in \rho(\pi)$ and $(s, s_2) \in \rho(\pi)$ implies $s_1 = s_2$ for all states $s, s_1, s_2$.*

### 9.7.1 Listing

**Theorem 9** *For arbitrary programs $\pi$, DL-formulas $F$ and $G$ the following formulas are tautologies. In any case it is assumed that the variable $x$ does not occur in $\pi$, i.e. $x \notin V^{\pi}$.*

1. $(\exists x\ \langle\pi\rangle F) \leftrightarrow (\langle\pi\rangle \exists x\ F)$
2. $(\forall x\ [\pi] F) \leftrightarrow ([\pi] \forall x\ F)$
3. $(\exists x\ [\pi] F) \rightarrow ([\pi] \exists x\ F)$
4. $([\pi] \exists x\ F) \rightarrow (\exists x\ [\pi] F)$ *under the precondition that $\pi$ is deterministic*
5. $(\langle\pi\rangle \forall x\ F) \rightarrow (\forall x\ \langle\pi\rangle F)$
6. $(\forall x\ \langle\pi\rangle F) \rightarrow (\langle\pi\rangle \forall x\ F)$ *under the precondition that $\pi$ is deterministic*
7. $(\langle\pi\rangle (F \wedge G)) \rightarrow ((\langle\pi\rangle F) \wedge \langle\pi\rangle G)$
8. $(\langle\pi\rangle (F \wedge G)) \leftrightarrow ((\langle\pi\rangle F) \wedge \langle\pi\rangle G)$ *under the precondition that no variable or non-rigid symbol occurring in $F$ occurs in $\pi$*

### 9.7.2 Proofs

We start with the following elementary observations on the semantics of $\mathcal{L}_r$-programs.

**Lemma 10** *For any program $\pi \in \Pi_r$ let $FV(\pi)$ be the set of variables occuring on the left hand side of an assignment statement in $\pi$ and $V^\pi$ all variables occuring in $\pi$.*

1. *The program $\pi$ only changes variables in $FV(\pi)$; that is, if $((\mathcal{M},\beta),(\mathcal{M}_1,\beta_1)) \in \rho(\pi)$ then $\beta(x) = \beta_1(x)$ for all variables $x \notin FV(\pi)$.*
2. *Variables outside $V^\pi$ do not influence the program $\pi$; that is, if if $x \notin V^\pi$ and $((\mathcal{A},\beta),(\mathcal{B},\gamma)) \in \rho(\pi)$. then also $((\mathcal{A},\beta^a_x),(\mathcal{B},\gamma^a_x)) \in \rho(\pi)$. for arbitrary $a$.*
3. *more general: If $((\mathcal{M},\beta),(\mathcal{M}_1,\beta_1)) \in \rho(\pi)$ and $\beta'$ is a variable assignment such that $\beta'(y) = \beta(y)$ for all $y \in V^\pi$ then there is $\beta'_1$ such that*
   (a) *$((\mathcal{M},\beta'),(\mathcal{M}_1,\beta'_1) \in \rho(\pi)$ and*
   (b) *$\beta'_1(x) = \beta'(x)$ for all $x \notin V^\pi$*
   (c) *$\beta'_1(y) = \beta_1(y)$ for all $y \in V^\pi$.*

We now turn to the proof of Theorem 9.

*(1).* Since this is the first proof in a series of similar spirit we present it in full detail.

The assumption $x \notin V^\pi$ will make the applications of the basic observation in the following proof possible.

$\Rightarrow$:

| | | | | |
|---|---|---|---|---|
| 1 | $(\mathcal{A},\beta)$ | $\models$ | $\exists x\langle\pi\rangle F$ | assumption |
| 2 | $(\mathcal{A},\beta^b_x)$ | $\models$ | $\langle\pi\rangle F$ | for some $b$ |
| 3 | $(\mathcal{B},\gamma)$ | $\models$ | $F$ | with $((\mathcal{A},\beta^b_x),(\mathcal{B},\gamma)) \in \rho(\pi)$ |
| 4 | $(\mathcal{B},\gamma)$ | $\models$ | $\exists x F$ | classical predicate logic |
| 5 | $(\mathcal{B},\gamma^{\beta(x)}_x)$ | $\models$ | $\exists x F$ | classical predicate logic |
| 6 | $(\mathcal{A},\beta)$ | $\models$ | $\langle\pi\rangle\exists x F$ | from $((\mathcal{A},\beta^b_x),(\mathcal{B},\gamma)) \in \rho(\pi)$ we get by our basic observation $((\mathcal{A},\beta),(\mathcal{B},\gamma^{\beta(x)}_x)) \in \rho(\pi)$ |

$\Leftarrow$:

| | | | | |
|---|---|---|---|---|
| 1 | $(\mathcal{A},\beta)$ | $\models$ | $\langle\pi\rangle\exists xF$ | assumption |
| 2 | $(\mathcal{B},\gamma)$ | $\models$ | $\exists xF$ | with $((\mathcal{A},\beta),(\mathcal{B},\gamma))\in\rho(\pi)$ |
| 3 | $(\mathcal{B},\gamma^b_x)$ | $\models$ | $F$ | for some $b$ |
| 4 | $(\mathcal{A},\beta^b_x)$ | $\models$ | $\langle\pi\rangle F$ | from $((\mathcal{A},\beta),(\mathcal{B},\gamma))\in\rho(\pi)$ we get by the basic observation $((\mathcal{A},\beta^b_x),(\mathcal{B},\gamma^b_x))\in\rho(\pi)$ |
| 5 | $(\mathcal{A},\beta)$ | $\models$ | $\exists x\langle\pi\rangle F$ | predicate logic |

*Proof of (2).* Follows from (1) and Exercise 9.18.3(1)

*Proof of (3).* Consider an arbitrary state $(\mathcal{A},\beta)$ in a Kripke structure $\mathcal{K}$.

| | | | | |
|---|---|---|---|---|
| 1 | $(\mathcal{A},\beta)$ | $\models$ | $\exists x\,[\pi]F$ | assumption |
| 2 | $(\mathcal{A},\beta^a_x)$ | $\models$ | $[\pi]F$ | for some $a$ |
| 3 | $(\mathcal{B},\gamma)$ | $\models$ | $F$ | for all $(\mathcal{B},\gamma)$with $((\mathcal{A},\beta^a_x),(\mathcal{B},\gamma))\in\rho(\pi)$ |
| 3 | $(\mathcal{B},\gamma)$ | $\models$ | $\exists xF$ | for all $(\mathcal{B},\gamma)$ with $((\mathcal{A},\beta^a_x),(\mathcal{B},\gamma))\in\rho(\pi)$ |
| 4 | $(\mathcal{A},\beta)$ | $\models$ | $[\pi]\exists xF$ | |

*Proof of (4).* Consider again an arbitrary state $(\mathcal{A},\beta)$ in a Kripke structure $\mathcal{K}$.

| | | | | |
|---|---|---|---|---|
| 1 | $(\mathcal{A},\beta)$ | $\models$ | $[\pi]\exists xF$ | assumption |
| 2 | $(\mathcal{B},\gamma)$ | $\models$ | $\exists xF$ | for all $(\mathcal{B},\gamma)$ with $((\mathcal{A},\beta),(\mathcal{B},\gamma))\in\rho(\pi)$ by determinacy this is equivalent to: |
| 3 | $(\mathcal{B},\gamma)$ | $\models$ | $\exists xF$ | for the unique $(\mathcal{B},\gamma)$ with $((\mathcal{A},\beta),(\mathcal{B},\gamma))\in\rho(\pi)$ |
| 4 | $(\mathcal{B},\gamma^b_x)$ | $\models$ | $F$ | for the unique $(\mathcal{B},\gamma)$ with $((\mathcal{A},\beta),(\mathcal{B},\gamma))\in\rho(\pi)$ and some $b\in B$ |
| 5 | $(\mathcal{A},\beta^b_x)$ | $\models$ | $[\pi]F$ | since by the basic observation $(\mathcal{B},\gamma^b_x)$ is the unique state with $((\mathcal{A},\beta^b_x),(\mathcal{B},\gamma^b_x))\in\rho(\pi)$ for some $b\in B=A$ |
| 5 | $(\mathcal{A},\beta^b_x)$ | $\models$ | $\exists x[\pi]F$ | predicate logic |
| 6 | $(\mathcal{A},\beta)$ | $\models$ | $\exists x[\pi]F$ | |

Notice, that the constant domain assumption is crucial for this argument.

*Proof of (5).* Follows from (3) and Exercise 9.18.3(1).

*Proof of (6).* Follows from (4) and Exercise 9.18.3(2).

*Proof of (7).*

| | | | | |
|---|---|---|---|---|
| 1 | $(\mathcal{A}, u\beta)$ | $\models$ | $\langle\pi\rangle(F \wedge G$ | assumption |
| 2 | $(\mathcal{B}, \gamma)$ | $\models$ | $F \wedge G$ | with $((\mathcal{A}, \beta), (\mathcal{B}, \gamma)) \in \rho(\pi)$ |
| 3 | $(\mathcal{B}, \gamma)$ | $\models$ | $F$ and $(\mathcal{B}, \gamma) \models G$ | |
| 4 | $(\mathcal{A}, \beta)$ | $\models$ | $\langle\pi\rangle F$ and $(\mathcal{A}, \beta) \models \langle\pi\rangle G$ | |
| 5 | $(\mathcal{A}, \beta)$ | $\models$ | $\langle\pi\rangle G \wedge \langle\pi\rangle F$ | |

*Proof of (8).* Because of (7) only the implication $\leftarrow$ needs to be proved.

| | | | | |
|---|---|---|---|---|
| 1 | $(\mathcal{A}, \beta)$ | $\models$ | $\langle\pi\rangle F \wedge \langle\pi\rangle G$ | assumption |
| 2 | $(\mathcal{B}_1, \gamma_1)$ | $\models$ | $F$ and $(\mathcal{B}_2, \gamma_2) \models G$ | with $((\mathcal{A}, \beta), (\mathcal{B}_i, \gamma_i)) \in \rho(\pi)$ |

Since $\gamma_1$ and $\gamma_2$ coincide on $x \notin V^\pi$ and no $x \in V^\pi$ occurs in $F$ we get

3 $(\mathcal{B}_1, \gamma_2) \models F$

Now, $f^{\mathcal{B}_1} = f^{\mathcal{B}_2}$ for all $f \in \Sigma_r \cup \Sigma_{nr} \setminus \Sigma^\pi_{nr} \cup \Sigma_{nr}$ and no $f \in \Sigma^\pi_{nr}$ occurs in $F$. Thus

4 $(\mathcal{B}_2, \gamma_2) \models F$

(2) and (4) yield $(\mathcal{B}_2, \gamma_2) \models F \wedge G$ and thus finally

5 $(\mathcal{A}, \beta) \models \langle\pi\rangle(F \wedge G)$

Notice again, that the argument would not work as presented without the constant domain assumption. We would have no guarantee that $\mathcal{B}_1$ and $\mathcal{B}_2$ share the same universe. What could $f^{\mathcal{B}_1} = f^{\mathcal{B}_2}$ mean is this context?

■

### 9.7.3 Comments

## 9.8 Conditional Terms

In this section we introduce a new syntactical concept into Dynamic Logic, that will later turn out to be helpful.

**Definition 31** *If $t_1$, $t_2$ are terms then*

$$\textit{if } u = w \textit{ then } t_1 \textit{ else } t_2$$

*is a conditional term.*

*In a state $(\mathcal{A}, \beta)$ conditional terms are interpreted as follows:*

$$(\textit{if } u = w \textit{ then } t_1 \textit{ else } t_2)^{(\mathcal{A},\beta)} = \begin{cases} t_1^{(\mathcal{A},\beta)} & \textit{if } \beta(u) = \beta(w) \\ t_2^{(\mathcal{A},\beta)} & \textit{otherwise} \end{cases}$$

As the next lemma shows, conditional terms in first-order formulas are a mere convenience.

**Lemma 11** *Let $\phi$ be a first-order formula and occ an occurence of the conditional term if $u = w$ then $t_1$ else $t_2$ in $\phi$. Let $\phi_1$ be obtained from $\phi$ by replacing occ by $t_1$ and $\phi_2$ by replacing occ by $t_2$. Then*

$$\phi \leftrightarrow (u = v \wedge \phi_1) \vee (u \neq v \wedge \phi_2)$$

**Proof:** Before we can start the proof of the lemma we state and prove a corresponding claim for term. Let $s$ be an arbitrary term in the extended syntax, *occ* and occurence of if $u = w$ then $t_1$ else $t_2$ in $t$. Let $s_1$, $s_2$ be the term arising from $s$ by replacing *occ* by $t_1$, $t_2$. then we get for any state $(\mathcal{A}, \beta)$

$$s^{(\mathcal{A},\beta)} = \begin{cases} s_1^{(\mathcal{A},\beta)} & \text{if} \quad \beta(u) = \beta(w) \\ s_2^{(\mathcal{A},\beta)} & \text{if} \quad \beta(u) \neq \beta(w) \end{cases}$$

The proof is by induction on the complexity of the term $s$. If $s$ is just a variable, then $s_1 = s_2 = s$, and the claim is obvious. If $s = f(s^1, \ldots, s^k)$ then $s_i = f(s_i^1, \ldots, s_i^k)$ for $i = 1$ or $i = 2$. Evaluating $s$ we get first $s^{(\mathcal{A},\beta)} = f^{\mathcal{A}}((s^1)^{(\mathcal{A},\beta)}, \ldots, (s^k)^{(\mathcal{A},\beta)})$ Using the induction hypothesis we conclude in case $\beta(u) = \beta(w)$ that $s^{(\mathcal{A},\beta)} = f^{\mathcal{A}}((s_1^1)^{(\mathcal{A},\beta)}, \ldots, (s_1^k)^{(\mathcal{A},\beta)})$. Where, as you may have guessed, $s_1^j$ arises from $s^j$ by replacing *occ* by $t_1$. The last equation now immediately yields $s^{(\mathcal{A},\beta)} = s_1^{(\mathcal{A},\beta)}$. If we use, likewise, $s_2^j$ to denote the term obtained from $s^j$ by replacing *occ* by $t_2$, we obtain in the case $\beta(u) \neq \beta(w)$: $s^{(\mathcal{A},\beta)} = f^{\mathcal{A}}((s_2^1)^{(\mathcal{A},\beta)}, \ldots, (s_2^k)^{(\mathcal{A},\beta)})$. Which again leads to the desired conclusion $s^{(\mathcal{A},\beta)} = s_2^{(\mathcal{A},\beta)}$.

It remains to consider the case $s = \text{if } x = y \text{ then } s_1 \text{ else } s_2$ The subcase, that $s$ is in fact the occurence *occ* to be replaced, and consequently $x \equiv u$, $y \equiv v$,and $s_i \equiv t_i$ is immediately obvious. It remains to consider the subcase that *occ* occurs in $s_1$ or $s_2$. Let us first take up the case $\beta(u) = \beta(v)$. This splits, naturally, again in two subcases. If $\beta(x) = \beta(y)$ then $s^{(\mathcal{A},\beta)} = s_1^{(\mathcal{A},\beta)}$ and induction hypothesis yields $s_1^{(\mathcal{A},\beta)} = s_{1,1}^{(\mathcal{A},\beta)}$. If $\beta(x) \neq \beta(y)$ then $s^{(\mathcal{A},\beta)} = s_2^{(\mathcal{A},\beta)}$ and the induction hypothesis helps us to get $s_2^{(\mathcal{A},\beta)} = s_{2,1}^{(\mathcal{A},\beta)}$. In total we have seen $s^{(\mathcal{A},\beta)} = s_1^{(\mathcal{A},\beta)}$.

In case $\beta(u) \neq \beta(v)$ we obtain $s^{(\mathcal{A},\beta)} = s_2^{(\mathcal{A},\beta)}$, completely analogous.

We are now ready to attack the proof of the lemma itself using structural induction on $\phi$. Furthermore, the best approach to prove for each step with this inductive proof that the given equivalence is true in any state $(\mathcal{A}, \beta)$, is to procced by case distinction wether $\beta(u) = \beta(v)$.

For the simplest case $\phi = r(s_1, \ldots, s_k)$ for a relation symbol $r$ we arrive at the proof obligations:

$$\begin{array}{ll} (\mathcal{A},\beta) \models r(s_1,\ldots,s_k) \leftrightarrow r(s_{1,1},\ldots,s_{k,1}) & \text{in case } \beta(u) = \beta(v) \\ (\mathcal{A},\beta) \models r(s_1,\ldots,s_k) \leftrightarrow r(s_{1,2},\ldots,s_{k,2}) & \text{in case } \beta(u) \neq \beta(v) \end{array}$$

By the first half of the proof we know

$$\begin{array}{ll} s_i^{(\mathcal{A},\beta)} = s_{i,1}^{(\mathcal{A},\beta)} & \text{in case } \beta(u) = \beta(v) \\ s_i^{(\mathcal{A},\beta)} = s_{i,2}^{(\mathcal{A},\beta)} & \text{in case } \beta(u) \neq \beta(v) \end{array}$$

which settles the case immediatedly.

If the induction hypothesis supplies us with $(\mathcal{A}, \beta) \models \phi_i \leftrightarrow \phi_{i,1}$ in case $\beta(u) = \beta(v)$ then we also get $(\mathcal{A}, \beta) \models \phi_1 \wedge \phi_2 \leftrightarrow \phi_{1,1} \wedge \phi_{2,1}$ and $(\mathcal{A}, \beta) \models \neg\phi_1 \leftrightarrow \neg\phi_{1,1}$ in case $\beta(u) = \beta(v)$. Likewise, for other propositional connectives and the other case $\beta(u) \neq \beta(v)$. As the last hurdle let us set out to prove $(\mathcal{A}, \beta) \models \forall x\phi_i \leftrightarrow (\forall x\phi_i)_1$ for the case $\beta(u) = \beta(v)$. This is equivalently transformed in to $(\mathcal{A}, \beta) \models \forall x\phi_i \leftrightarrow (\forall x\phi_{i,1})$. Now, it turns out that the even stronger claim $(\mathcal{A}, \beta) \models \forall x(\phi_i \leftrightarrow \phi_{i,1})$ can be shown to be true, simply because $\mathcal{A}, \beta) \models (\phi_i \leftrightarrow \phi_{i,1}$ is, by induction hyppothesis, true for all states $(\mathcal{A}, \beta)$.

∎

## 9.9 Substitutions

### 9.9.1 Retrospective

Before we approach the particularities of substitutions in formulas of Dynamic Logic we look at the crucial difficulties of substitutions in first-order logic.

Let $\mathbb{N}$ be the well acquainted structure of the natural numbers and $A$ the formula $z \geq 2 \wedge \forall x(div(x,z) \rightarrow (x = 0 \vee x = 1 \vee x = z)$. Then $(\mathbb{N}, \beta) \models A$ if $\beta(z)$ is a prime number. Let $A'$ be the formula obtained from $A$ by substituting the term $t = 2^z - 1$ for $z$, in symbols $A' = A(t/z)$. Then $(\mathbb{N}, \beta) \models A'$ if and only if $t^{(\mathbb{N},\beta)}=2^{\beta(z)} - 1$ is a prime number. If we substitute the term $s = 2 * x$ for $z$ we get $A'' = 2 * x \geq 2 \wedge \forall x(div(x, 2 * x) \rightarrow (x = 0 \vee x = 1 \vee x = 2 * x)$. Obviously $(\mathbb{N}, \beta) \models A''$ is always false.

The first substitution somehow preserves the meaning of the formula $A$ while the second does not. The informal notion of a substitution $\sigma$ *preserving* the meaning of the formula $A$ is made precise by requiring that the substitution principle holds true.

**Definition 32** *For a substitution $\sigma$ and a formula $A$ the* substitution principle *requires that for all states $(\mathcal{M}, \beta)$*

$$(\mathcal{M}, \beta) \models \sigma(A) \text{ iff } (\mathcal{M}, \beta') \models A$$

*with $\beta'(x) = \sigma(x)^{(\mathcal{M},\beta)}$ for all variables $x$.*

**Definition 33** *A substitution $\sigma$ is* allowed *for a formula $A$ if for all free variable occurrences $x$ in $A$ and all variables $z$ in $\sigma(x)$ the occurrence $x$ is not within the scope of a quantifier binding $z$.*

The following lemma is a well-known result of first-order logic. Its proof is tedious but straight forward.

**Lemma 12 (Substitution Lemma for First-Order Logic)**
*If $A$ is a formula of first-order logic and $\sigma$ an allowed substitution for $A$, then the substitution principle holds true.*

### 9.9.2 Substitutions in Dynamic Logic

There are limits to possible extensions of Lemma 12 to Dynamic Logic. If a program $\pi$, when called in state $(\mathcal{M}, \beta)$, does not only change the values of the program variables given by $\beta$, but also changes $\mathcal{M}$, then there can obviously be no straightforward analgon of Lemma 12 for formulas of the form $\langle\pi\rangle A$ or $[\pi]A$. On the other hand, the substitution lemma is needed to prove correctness of the axiom handling the assignment rule (see Section 9.13). We will come up with a new approach, introducing the concept of *updates*, later to solve this problem. Since updates will provide the general solution the rest of this section may seem outdated. We chose, nevertheless, to include it here since it may be useful to avoid updates when they are not needed.

Let us look at a few examples in the logic $\mathcal{L}_r$.

**Example 1**

$$(\mathbb{N}, \beta) \models \langle x = x + y\rangle even(x)$$

This statement is true if $\beta(x)$ and $\beta(y)$ are both even or both uneven. If $\sigma$ substitutes $z + u$ for $y$ then

$$(\mathbb{N}, \beta) \models \langle x = x + \sigma(y)\rangle even(x)$$

is true if and only if $\beta(x)$ and $\beta(z) + \beta(u)$ are both even or both uneven. That sounds reasonable.

If on the other hand we have $\sigma(y) = x$ then

$$(\mathbb{N}, \beta) \models \langle x = x + \sigma(y)\rangle even(x)$$

is always true, which does not sound right.

We may draw from this example the conclusion that whenever we substitute a term $t$ for a variable on the left hand side of an assignment $x = s$, then $x$ should not occur in $t$. In this way program variables $x$ on the left hand side of an assignment statement play a similar role as quantifiers expressions $\forall z$ or $\exists z$.

**Example 2** For a substitution $\sigma$ with $\sigma(y) = x$ the claim $(\mathbb{N}, \beta) \models \sigma(\langle x = 5\rangle x = y)$ amounts to

$$(\mathbb{N}, \beta) \models \langle x = 5\rangle x = x$$

which is of course always true, regardless of $\beta$. On the other hand

$$(\mathbb{N}, \beta') \models \langle x = 5\rangle x = y$$

is only true if $\beta'(y) = 5$. This shows that the *binding force* of the variable $x$ in the assignment statement extends also to the formula following this statement.

**Definition 34** *A substitution $\sigma$ is* allowed *for a $\mathcal{L}_r$-formula $A$, (a program $\pi$), if*

1. *for all $x$ occuring in $\pi$ as left hand side of an assignment statement $\sigma(x) = x$,*
2. *for all free variable occurrences $y$ in $A$ (in $\pi$) and all variables $x$ in $\sigma(y)$ the occurrence $y$*
   (a) *is not within the scope of a quantifier $\forall x$ or $\exists x$, and*
   (b) *$x$ does not occur as a left-hand side of an assignment*

**Lemma 13 (Substitution Lemma for Rigid Dynamic Logic)**

1. *If $\pi$ is in $\Pi_r$ and $\sigma$ a substitution allowed for $\pi$, then the program substitution principle holds true, i.e.*

   (a)
   $$((\mathcal{M}, \beta), (\mathcal{M}, \beta_1)) \in \rho(\sigma(\pi))$$
   *implies*
   $$((\mathcal{M}, \beta'), (\mathcal{M}, \beta'_1)) \in \rho(\pi)$$
   *and*

   (b)
   $$((\mathcal{M}, \beta'), (\mathcal{M}, \beta_2)) \in \rho(\pi)$$
   *implies the existence of $\beta_1$ with*
   $$((\mathcal{M}, \beta), (\mathcal{M}, \beta_1)) \in \rho(\sigma(\pi)) \text{ and } \beta'_1 = \beta_2$$
   *with $\beta'(x) = \sigma(x)^{(\mathcal{M},\beta)}$ and $\beta'_1(x) = \sigma(x)^{(\mathcal{M},\beta_1)}$*

2. *If $A$ is a $\mathcal{L}_r$-formula and $\sigma$ a substitution allowed for $A$, then the substitution principle holds true, i.e.*

$$(\mathcal{M}, \beta) \models \sigma(A) \Leftrightarrow (\mathcal{M}, \beta') \models A$$

*with $\beta'(x)$ as above.*

### 9.9.3 Proofs

The proof of Lemma 13 proceeds by simultaneous structural induction on $\pi$ and $A$.

**Proof of Part 1** Since we are dealing with a rigid Dynamic Logic we will fix an arbitrary structure $\mathcal{M}$ and write throughout this proof $(\beta, \gamma) \in \rho(\pi)$ instead of $((\mathcal{M}, \beta), (\mathcal{M}, \gamma)) \in \rho(\pi)$.

**Assignment** So let $\sigma$ be an allowed substitution for $\pi \equiv x = t$ and consider $(\beta, \beta_1) \in \rho(\sigma(x = t)) = \rho(x = \sigma(t))$. Thus

$$\beta_1(y) = \begin{cases} \beta(y) & \text{if } y \neq x \\ \sigma(t)^{(\mathcal{M},\beta)} & \text{if } y = x \end{cases}$$

Let $\beta_2'$ be the variable assignment satisfying $(\beta', \beta_2') \in \rho(x = t)$. Thus

$$\beta_2'(y) = \begin{cases} \beta'(y) & \text{if } y \neq x \\ t^{(\mathcal{M},\beta')} & \text{if } y = x \end{cases}$$

It remains to show

$$\beta_1' = \beta_2'$$

For $y \neq x$ we get

$$\beta_1'(y) = \sigma(y)^{(\mathcal{M},\beta_1)} = \sigma(y)^{(\mathcal{M},\beta)}$$

since $\beta_1$ and $\beta$ differ only on the variable $x$, and $x$ does not occur in any $\sigma(y)$ with $\sigma(y) \neq y$. For $y \neq x$ we also have

$$\beta_2'(y) = \beta'(y) = \sigma(y)^{(\mathcal{M},\beta)}$$

and thus $\beta_1'(y) = \beta_2'(y)$, as desired. Now let us look at the case $y = x$.

$$\beta_1'(x) = \sigma(x)^{(\mathcal{M},\beta_1)} = x^{(\mathcal{M},\beta_1)} = \beta_1(x) = \sigma(t)^{(\mathcal{M},\beta)}$$

On the other hand

$$\beta_2'(x) = t^{(\mathcal{M},\beta')}$$

By the subsitution principle for terms we find the last missing link $t^{(\mathcal{M},\beta')} = \sigma(t)^{(\mathcal{M},\beta)}$

We have so far proved

$$(\beta, \beta_1) \in \rho(\sigma(x = t)) \Rightarrow (\beta', \beta_1') \in \rho(x = t)$$

The remaining part is trivial since $x = t$ is a deterministic and terminating program.

**Composition** If $(\beta, \beta_1) \in \rho(\sigma(\pi_1; \pi_2)) = \rho(\sigma(\pi_1); \sigma(\pi_2))$ then there is $\gamma$ with $(\beta, \gamma) \in \rho(\sigma(\pi_1))$ and $(\gamma, \beta_1) \in \rho(\sigma(\pi_2))$. By induction hypothesis we obtain $(\beta', \gamma') \in \rho(\pi_1)$ and $(\gamma', \beta_1') \in \rho(\pi_2)$. From this $(\beta', \beta_1') \in \rho(\pi_1; \pi_2)$, as desired.

Now for the second part. Consider $(\beta', \beta_2) \in \rho(\pi_1; \pi_2)$. Thus for some $\delta$ we have $(\beta', \delta) \in \rho(\pi_1)$ and $(\delta, \beta_2) \in \rho(\pi_2)$. By induction hypothesis there exists $\gamma$ such that $(\beta, \gamma) \in \rho(\sigma(\pi_1))$ and $\gamma' = \delta$. Thus we have $(\gamma', \beta_2) \in \rho(\pi_2)$ and a second application of the induction hypothesis yields $\beta_1$ with $(\gamma, \beta_1) \in \rho(\sigma(\pi_2))$ and $\beta_1' = \beta_2$. This finishes the argument since we already have $(\beta, \beta_1) \in \rho(\sigma(\pi_1; \pi_2))$.

**Branching** part (a) The assumption

$$\begin{array}{rcl} (\beta, \gamma) & \in & \rho(\sigma(\text{if } (con) \ \{\pi_1\} \ \text{else } \{\pi_2\})) \\ & = & \rho(\text{if } \ \sigma(con) \ \{\sigma(\pi_1)\} \ \text{else } \{\sigma(\pi_2)\})) \end{array}$$

implies by definition that either

$(\mathcal{M}, \beta) \models \sigma(con)$ and $(\beta, \gamma) \in \rho(\sigma(\pi_1))$

or $(\mathcal{M}, \beta) \models \neg\sigma(con)$ and $(\beta, \gamma) \in \rho(\sigma(\pi_2))$

By induction hypothesis, from part 2, $(\mathcal{M}, \beta) \models \sigma(con)$ is equivalent to $(\mathcal{M}, \beta') \models con$. By induction hypothesis used on $\pi_i$ we have $(\beta', \gamma') \in \rho(\pi_1)$ for the first alternative and $(\beta', \gamma') \in \rho(\pi_2)$ for the second. Taken together we obtain $(\beta', \gamma') \in \rho(\text{if } (con) \ \{\pi_1\} \ \text{else } \{\pi_2\})$.

Part (b). Assume $((\mathcal{M}, \beta'), (\mathcal{M}, \beta_2)) \in \rho(\text{if } (con) \ \{\pi_1\} \ \text{else } \{\pi_2\})$. There are, obviously, two cases

1. $(\mathcal{M}, \beta') \models con$ and $((\mathcal{M}, \beta'), (\mathcal{M}, \beta_2)) \in \rho(\pi_1)$
   or
2. $(\mathcal{M}, \beta') \models \neg con$ and $((\mathcal{M}, \beta'), (\mathcal{M}, \beta_2)) \in \rho(\pi_2)$

Assume case 1 applies. Then we get by using the induction hypothesis for the program $\pi_1$ a subsitution $\beta_1$ such that

$$(\beta, \beta_1) \in \rho(\sigma(\pi_1))$$

and $\beta_1' = \beta_2$. Using the induction hypothesis of part 2 of the lemma on the formula *con*, we obtain

$$(\mathcal{M}, \beta') \models con \Leftrightarrow (\mathcal{M}, \beta) \models \sigma(con)$$

Taken together these two statements yield

$$(\beta, \beta_1) \in \rho(\text{if } (con)\ \{\pi_1\}\ \text{ else } \{\pi_2\})$$

The case 2 is, of course, handled in the very same way.

**While Loop** There are no surprises in this part of the proof. The only thing to point out is that we need the induction hypothesis of both parts, program part and formula part, simultaneously. But, let us quickly run through the details. We consider the program $\pi = \text{while } (con)\ \{\pi_0\})$.
part (a)
We need to prove that $(\beta, \beta_1) \in \rho(\sigma(\pi))$ implies $(\beta', \beta_1') \in \rho(\pi)$. The assumption provides us with a finite sequence $\gamma_1, \ldots, \gamma_n$ of states such that $\gamma_1 = \beta$, $\gamma_n = \beta_1$ and $(\gamma_i, \gamma_{i+1}) \in \rho(\sigma(\pi_0))$ and $\gamma_i \models \sigma(con)$ for $1 \leq i < n$ and $\gamma_n \models \neg\sigma(con)$. By induction hypothesis we get for all $1 \leq i < n$ $(\gamma_i', \gamma_{i+1}') \in \rho(\pi_0)$, $\gamma_i' \models con$ and also $\gamma_n' \models \neg con$. By definition this is $(\gamma_1', \gamma_n') \in \rho(\text{while } (con)\ \{\pi_0\}) = \rho(\pi)$.

part (b)) Let $\beta$ be as in part (a) and assume $(\beta', \beta_2) \in \rho(\pi)$ for some $\beta_2$. We want to show the existence of a state $\beta_1$ with $(\beta, \beta_1) \in \rho(\sigma(\pi))$ and $\beta_1' = \beta_2$. Unwinding the semantics of the while loop we obtain from our assumption states $\delta_0, \ldots, \delta_k$ with $\delta_0 = \beta'$, $\delta_k = \beta_2$, $\delta_k = \beta_2$, for all $1 \leq i < k$ $(\delta_i, \delta_{i+1}) \in \rho(\pi_0)$, $\delta_i \models con$ and also $\delta_k \models \neg con$. By induction hypthesis we obtain successively states $\delta^i$ such that $\delta^0 = \beta$, and $(\delta^i, \delta^{i+1}) \in \rho(\sigma(\pi_0))$ for $0 \leq i < k$ and $(\delta^i)' = \delta_i$ for all $1 \leq i \leq k$. Note, that this means in particular $(\delta^k)' = \beta_2$. We also obtain from $\delta_i = (\delta^i)' \models con$ by induction hypothesis $\delta^i \models \sigma(con)$ for all $1 \leq i < k$ and also $\delta^k \models \neg\sigma(con)$. Taken alltogether this is $(\beta, \delta^k) \in \rho(\sigma(\pi))$ and $(\delta^k)' = \beta_2$.

**Proof of Part 2** To simplify notation we will for any variable assignment $\gamma$ denote by $\gamma'$ the usual function $\gamma'(x) = \sigma(x)^{(\mathcal{M},\gamma)}$, where $\sigma$ and $\mathcal{M}$ are determined by the context.

**Diamond** Assume we already know that the substitution principle holds true for the program $\pi$ and the formula $A$ and we want to prove it for $\langle\pi\rangle A$.

From $(\mathcal{M},\beta) \models \sigma(\langle\pi\rangle A)$ we infer the existence of a variable assignment $\beta_1$ with

$$((\mathcal{M},\beta),(\mathcal{M},\beta_1)) \in \rho(\sigma(\pi))$$
$$(\mathcal{M},\beta_1) \models \sigma(A)$$

By induction hypothesis

$$((\mathcal{M},\beta'),(\mathcal{M},\beta_1') \in \rho(\pi)$$
$$(\mathcal{M},\beta_1') \models A$$

Which immediately yields $(\mathcal{M},\beta') \models \langle\pi\rangle A$.

If we start from $(\mathcal{M},\beta') \models \langle\pi\rangle A$ we obtain first the existence of $\beta_2$ such that

$$((\mathcal{M},\beta'),(\mathcal{M},\beta_2) \in \rho(\pi)$$
$$(\mathcal{M},\beta_2) \models A$$

Since $\pi$ by induction hypothesis satisfies the substitution principle there is $\beta_1$ with

$$((\mathcal{M},\beta),(\mathcal{M},\beta_1) \in \rho(\sigma(\pi))$$
$$\beta 1' = \beta_2$$

The last equation now yields $(\mathcal{M},\beta 1') \models A$ which in turn gives, using the induction hypothesis for $A$, $(\mathcal{M},\beta 1) \models \sigma(A)$, Altogether, $(\mathcal{M},\beta) \models \sigma(\langle\pi\rangle A)$, as needed.

**Box** We first assume $(\mathcal{M},\beta) \models \sigma([\pi]A)$ with the aim to prove $(\mathcal{M},\beta') \models [\pi]A$.

For any $\beta_2$ with $((\mathcal{M},\beta'),(\mathcal{M},\beta_2) \in \rho(\pi)$ we want to show $(\mathcal{M},\beta_2) \models A$. An appeal to the induction hypothesis supplies us with $\beta_1$ satisfying

$$((\mathcal{M},\beta),(\mathcal{M},\beta_1)) \in \rho(\sigma(\pi))$$
$$\text{and } \beta_1' = \beta_2$$

By our case assumption we obtain $(\mathcal{M}, \beta_1) \models \sigma(A)$, which by induction hypothesis, this time applied to the formula $A$, yields $(\mathcal{M}, \beta_1') \models A$. This is the same as $(\mathcal{M}, \beta_2) \models A$ and we have, in fact, proved $(\mathcal{M}, \beta') \models [\pi]A$.

Now, let us inversely assume $(\mathcal{M}, \beta') \models [\pi]A$ and try to infer $(\mathcal{M}, \beta) \models \sigma([\pi]A)$. To this purpose, consider $\beta_1$ with $((\mathcal{M}, \beta), (\mathcal{M}, \beta_1)) \in \rho(\sigma(\pi))$. By induction hypothesis on $\pi$ this implies $((\mathcal{M}, \beta'), (\mathcal{M}, \beta_1')) \in \rho(\pi)$. By case assumption we obtain $(\mathcal{M}, \beta_1') \models A$. Applying the induction hypothesis on $A$ we obtain $(\mathcal{M}, \beta_1) \models \sigma(A)$. Altogether, this finishes the proof of $(\mathcal{M}, \beta) \models \sigma([\pi]A)$.

The remaining cases of Part 2 are as in first-order logic.

∎

### 9.9.4 Comments

The program substitution principle formulated as part of Lemma 13 is more general than would have been necessary for $\Pi_r$, since it also covers non-deterministic programs. For deterministic program the substitution principle is equivalent to the first part of item 1 in 13, i.e.

$$((\mathcal{M}, \beta), (\mathcal{M}, \beta_1)) \in \rho(\sigma(\pi)) \Rightarrow ((\mathcal{M}, \beta'), (\mathcal{M}, \beta_1')) \in \rho(\pi)$$

plus

$$\rho(\pi) \neq \emptyset \Rightarrow \rho(\sigma(\pi)) \neq \emptyset$$

It is apparent from Definition 34 that variables occuring at least once at the left side of an assignment behave differently from the remaining variables. This suggest separating the set $V$ of variables in *program variables*, in the set $V_{prog}$ and *logical variables* in $V_{log}$

- program variables cannot be quantified, for any substitution $\sigma$ and program variable $x$ we have $\sigma(x) = x$, i.e. program variabes cannot be substituted, and $x$ does not occur in $\sigma(y)$ for logical variables $y$. Of course, program variables may be changed by program execution.
- logical variables never occur as the lefthand side of an assignment, are implicitely universally quantified. Logical variables are *rigid* they cannot be changed by program execution.

By indroducing new program variables, if necessary, every program $\pi$ can be equivalently transformed into $\langle x_1 = v_1; \ldots; x_n = v_k; \pi_0 \rangle$, with $x_i$ program variables and $v_i$ logical variables such that no logical variable occurs in $\pi_0$.

## 9.10 Arrays

### 9.10.1 Example

$$\begin{array}{l} for\ (int\ a = \ p; a < l - 1; a = a + 1) \\ \quad \{seq[a] = seq[a + 1];\} \end{array}$$

Figure 9.3: The Program Snippet Using Arrays

A number of papers have been published on how to treat arrays in Hoare Logic or Dynamic Logic, see e.g. [Apt, 1981]. The usual approach tried to stay as close as possible to the initial set-up of these logics. We took (in Section 9.3) a more radical step by indroducing *non-rigid* functions.

Consider the code snippet in Figure 9.3. To model its semantics we would have, among other, (program) variables $p$, $a$ and a non-rigid function symbol $seq$. If $\pi$ consists of the single statement $seq[a] = seq[a + 1];$,then

$$((\mathcal{M}, \beta), (\mathcal{M}_1, \beta)) \in \rho(\pi)$$

if $\mathcal{M}_1$ coincides with $\mathcal{M}$ except for the interpretation of the function symbol $seq$ which is given by

$$seq^{\mathcal{M}_1}(x) = \begin{cases} seq^{\mathcal{M}}(x) & \text{if } x \neq \beta(a) \\ seq^{\mathcal{M}}(x + 1) & \text{if } x = \beta(a) \end{cases}$$

## 9.11 Generalized Substitutions

### 9.11.1 Motivation

So far we have only considered substitutions for variables. Is it possible to define a more general notion that substitutes one term for another term? Let

us assume we want to define a generalized substitution $\sigma$ that replaces $f(2)$ by 5. What should be

1. $\sigma(g(f(2))$?
2. $\sigma(g(f(x))$?
3. $\sigma(f(f(x))$?

The first example is easy: $\sigma(g(f(2)) = g(5)$. The second example takes a moment's reflection. Then the conditional terms from Section 9.8 come in handy: $\sigma(g(f(x)) = \text{if } x = 2 \text{ then } g(5) \text{ else } g(f(x))$. Here we need to make a decision. Do we want to replace one possible occurence of $f(2)$ in $f(f(x))$ or all? The unanimously adopted solution (see e.g. [Apt & Olderog, 1991][Section 2.6]) is to replace all possible occurences.

`to be reconsidered`

In our example we would thus obtain

$$\sigma(f(f(x))) = \text{if (if } x = 2 \text{ then } 5 \text{ else } f(x)) = 2 \text{ then } 5 \text{ else } f(f(x))$$

This we could simplify to

$$\sigma(f(f(x))) = \text{if } (x \neq 2 \wedge f(x) = 2) \text{ then } 5 \text{ else } f(f(x))$$

This is quite different from simple texual replacements. The main reason why this is the right generalization is that it makes the following generalized substitution lemma, Lemma 14, true.

## 9.11.2 Definition

**Definition 35 (Generalized Substitution)**

`to be reconsidered`

*Let $\sigma$ be the generalized substitution that substitutes $t$ for $f(t_1, \ldots, t_n)$ where $t, t_1, \ldots, t_n$ are arbitrary terms. We define for an arbitrary term $s$ the result $\sigma(s)$ of applying $\sigma$ to $s$.*

$$\sigma(s) = \begin{cases} x & \text{if } \; s = x \text{ a variable} \\ s & \text{if } \; s = g(s_1, \ldots, s_k) \text{ with } g \neq f \\ \text{if } (\bigwedge_{i=1}^{n} \sigma(s_i) = t_i) \text{ then } t \text{ else } f(\sigma(s_1), \ldots, \sigma(s_n)) & \\ & \text{if } \; s = f(s_1, \ldots, s_n) \end{cases}$$

**Lemma 14 (Generalizied Substitution Lemma)** *Let $\sigma$ be the generalized substitution that replaces $f(t_1, \ldots, t_n)$ by $t$. Let $S = (\mathcal{M}, \beta)$, $S_1 = (\mathcal{M}_1, \beta)$ be states such that $\mathcal{M}_1$ coincides with $\mathcal{M}$ with the exception of the interpretation of $f$:*

$$f^{\mathcal{M}_1}(d_1, \ldots, d_n) = \begin{cases} t^S & \text{if } d_i = t_i^S \text{ for all } 1 \leq i \leq n \\ f^{\mathcal{M}}(d_1, \ldots, d_n) & \text{otherwise} \end{cases}$$

*Then for all terms $s$*

$$s^{S_1} = (\sigma(s))^S$$

Note that $S_1 = (\mathcal{M}_1, \beta)$ is the state reached by executing the assignment $f(t_1, \ldots, t_n) = t$ in state $S = (\mathcal{M}, \beta)$. Also note that $\beta$ is not changed.

**Proof:** The proof proceeds, as you would have expected, by induction on the complexity of the term $s$.
The only interesting case is $s = f(s_1, \ldots, s_n)$. In this case we get

$$\begin{aligned} s^{S_1} &= f^{\mathcal{M}_1}(d_1, \ldots, d_n) \text{ with } d_i = s_i^{S_1} \text{ for all } 1 \leq i \leq n \\ \sigma(s)^S &= (\text{if } (\textstyle\bigwedge_{i=1}^n \sigma(s_i) = t_i) \text{ then } t \text{ else } f(\sigma(s_1), \ldots, \sigma(s_n)))^S \end{aligned}$$

**Case 1** $s_i^{S_1} = t_i^S$ for all $1 \leq i \leq n$
By definition of $f^{S_1}$ we get immediately $s^{S_1} = t^S$. Furthermore the induction hypothesis yields $s_i^{S_1} = \sigma(s_i)^S = t_i^S$ for all $1 \leq i \leq n$ and thus the conditional term also evaluates to $\sigma(s)^S = t^S$ and we are finished.

**Case 2** otherwise
By definition of $f^{S_1}$ we get $s^{S_1} = f^M(s_1^{S_1}, \ldots, s_n^{S_n})$ which by induction hypothesis also yields

$$\begin{aligned} s^{S_1} &= f^M((\sigma(s_1)^S, \ldots, (\sigma(s_n)^S)) \\ &= f(\sigma(s_1), \ldots, \sigma(s_n))^S \end{aligned}$$

But, also the conditional term leads to the same end result: $\sigma(s)^S = f(\sigma(s_1), \ldots, \sigma(s_n))^S$

∎

## 9.12 Sequent Calculus

We assume that the reader has some familarity with the sequent calculus. Extensive accounts may be found e.g. in [Gallier, 1986, Sperschneider & Antoniou, 1991] or in the lecture notes [Menzel & Schmitt, 2001]. We quickly review here what will be needed for the following sections.

### 9.12.1 Sequent Rules

**Definition 36**

1. *A sequent is of the form*

$$\Gamma \rightarrowtail \Delta$$

*where* $\Gamma$ *and* $\Delta$ *are sets of formulas. Traditionally, the part to the left of the sequent arrow, i.e.* $\Gamma$ *in our example, is called the* antecedent *and the right part, i.e.* $\Delta$ *in our example, is called the* succedent *of the sequent.*

2. *Let* $\mathcal{K} = (S, \rho)$ *be a DL-Kripke structure,* $(\mathcal{A}, \beta)$ *a state in* $S$*. A sequent* $\Gamma \rightarrowtail \Delta$ *is true in* $\mathcal{K}$ *for state* $(\mathcal{A}, \beta)$*, in symbols* $(\mathcal{A}, \beta) \models \Gamma \rightarrowtail \Delta$ *iff*

$$(\mathcal{A}, \beta) \models \bigwedge \Gamma \rightarrow \bigvee \Delta$$

3. *A sequent* $\Gamma \rightarrowtail \Delta$ *is true in* $\mathcal{K}$*, in symbols* $\mathcal{K} \models \Gamma \rightarrowtail \Delta$ *iff* $(\mathcal{A}, \beta) \models \bigwedge \Gamma \rightarrow \bigvee \Delta$ *for all* $(\mathcal{A}, \beta) \in S$*.*

4. *A sequent* $\Gamma \rightarrowtail \Delta$ *is called* universally valid *if* $\mathcal{K} \models \Gamma \rightarrowtail \Delta$ *holds for all Kripke structures* $\mathcal{K}$ *in the signature of the sequent.*

**Definition 37**

1. *A sequent rule is of the form*

$$\frac{\Gamma_1 \rightarrowtail \Delta_1}{\Gamma_2 \rightarrowtail \Delta_2} \text{ or } \frac{\Gamma_1 \rightarrowtail \Delta_1 \quad \Gamma_1' \rightarrowtail \Delta_1'}{\Gamma_2 \rightarrowtail \Delta_2}$$

$\Gamma_1 \rightarrowtail \Delta_1$ *and* $\Gamma_1' \rightarrowtail \Delta_1'$ *are called the premise(s) of the rule and* $\Gamma_2 \rightarrowtail \Delta_2$ *is called the conclusion.*

2. *A sequent rule*

$$\frac{\Gamma_1 \rightarrow \Delta_1 \quad \Gamma_1' \rightarrow \Delta_1'}{\Gamma_2 \rightarrow \Delta_2}$$

*is* sound *if* $\Gamma_2 \rightarrow \Delta_2$ *is universally valid whenever* $\Gamma_1 \rightarrow \Delta_1$ *and* $\Gamma_1' \rightarrow \Delta_1'$ *are universally valid.*

Here are some examples of sound sequent rules.

1. $\dfrac{}{\Gamma^1, A, \Gamma^1 \rightarrow \Delta^1, A, \Delta^2}$

2. $\dfrac{\Gamma, \Gamma' \rightarrow \Delta, A, \Delta'}{\Gamma, \neg A, \Gamma' \rightarrow \Delta, \Delta'}$

3. $\dfrac{\Gamma \rightarrow \Delta, A(y/x), \Delta'}{\Gamma \rightarrow \Delta, \forall x A, \Delta'}$
   where $y$ has no free occurrence in a formula in $\Gamma, \Delta, \Delta'$

4. $\dfrac{\Gamma, A, \Gamma' \rightarrow \Delta \quad \Gamma, \Gamma' \rightarrow \Delta, A, \Delta'}{\Gamma, \Gamma' \rightarrow \Delta, \Delta'}$

In Chapter 15, the Appendix on Axiom Systems of the Sequent Calculus, lists two complete axiom systems for (one-sorted) predicate calculus, $S_0$ and $S_0^{fv}$.

### 9.12.2 Proof Trees

The rules of a sequent calculus, like $S_0$ e.g., are used to built proof trees. A proof tree is a tree whose nodes are labeled by sequents. Suppose, we want to prove that a formula $F$ is a tautology. Then we start the proof with the root node $\rightarrow F$. Using the rules from bottom to top we built proof trees as e.g. shown in Figures 9.4 and 9.5. The rules with two premises yield the branching of the proof tree. In the rule system $S_0$, see Chapter 15, a proof tree is called *closed* if every leaf node is labeled by an axiom. We will also speak of *branches* of a proof tree in the usual sense. A branch is called closed if its leaf node is labelled by an axiom, otherwise it is called open. Thus in $S_0$ a proof tree is closed iff all its branches are closed.

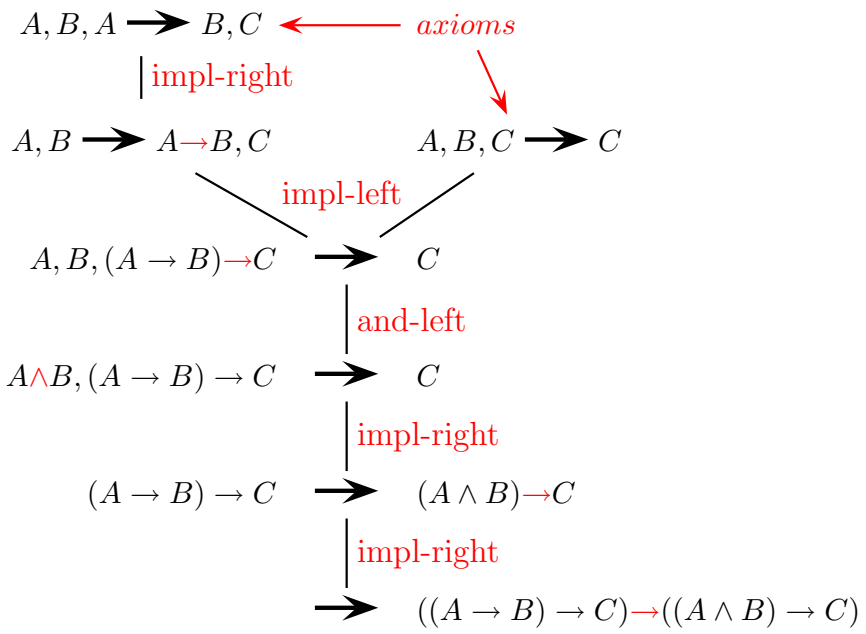



Figure 9.4: Example of a closed proof tree

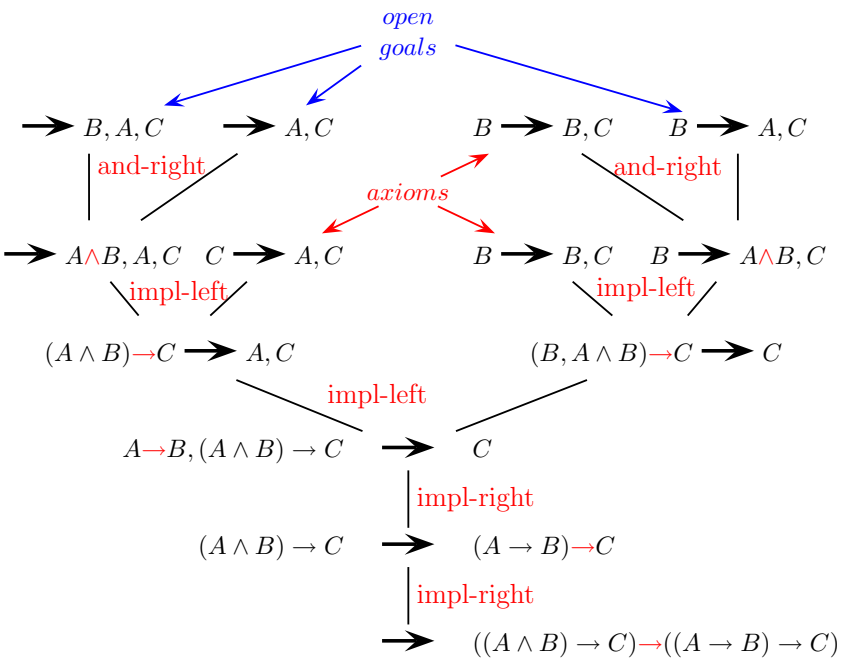
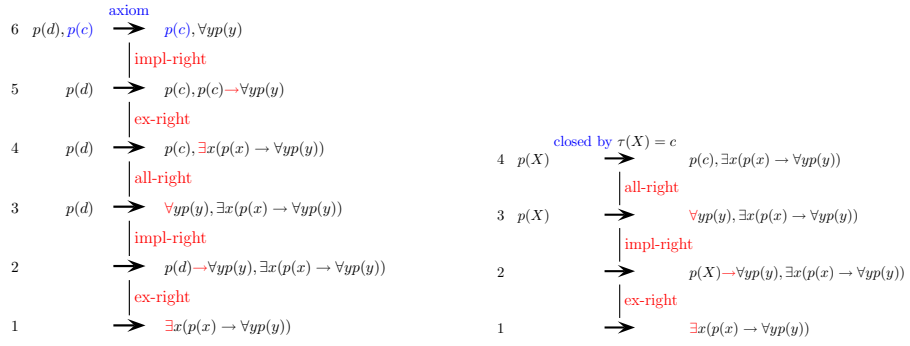



Figure 9.5: Example of an open proof tree

A proof tree $T$ in the system $S_0^{fv}$ is closed if there is a substitution $\tau$ of the free variables in $T$ by ground terms such that every branch of the proof tree $\tau(T)$ is closed.

**Theorem 15** *Let $S = \Gamma \rightarrow \Delta$ be a sequent containing only formulas of first-order predicate logic without free variables.*

*Then $S$ is a tautology iff there is a closed proof tree with root label $S$.*




(a) in $S_0$ (b) in $S_0^{fv}$

Figure 9.6: Proof of $\exists x(p(x) \rightarrow \forall y p(y)))$

### 9.12.3 Comments

Note, that in our approach antecedents and succedents are sets of formulas. This is in contrast with many other versions of sequent calculi, where sequences or multi-sets are used instead. Treating antecedents and succedents as sets simplifies the rules; we do not need rules re-ordering formulas in a sequent or omitting duplicate occurrences. On the other hand the implementation of sets is less straight forward.

In a sequent $\Gamma \rightarrow \Delta$ the sets $\Gamma$ and $\Delta$ may be empty, even both at the same time. Truth and validity is in this case determined by the (usual) stipulation that an empty disjunction is false and an empty conjunction is true. The sequent $\emptyset \rightarrow \emptyset$ is thus logically equivalent to **true** $\rightarrow$ **false** and thus evaluates to **false** in any state of any Kripke structure. Empty sets are usually omitted. Thus we write $\rightarrow$ instead of $\emptyset \rightarrow \emptyset$.

The rules themselves do not suggest any preference on how to built proof trees. One could, following a mathematically oriented method, start from axioms and apply proof rules from top to bottom until the desired root sequent is reached. Alternatively, one could start from the sequent to be proved and apply proof rules from bottom to top until every branch ends in an axiom. We prefer the second, goal oriented view. This is the reason, why we call sequents in a proof *goals*.

## 9.13 The Assignment Rule

### 9.13.1 The Rule

$$\frac{\Gamma(z/x), x \doteq^{s} t(z/x) \rightarrow F, \Delta(z/x)}{\Gamma \rightarrow \langle x = t \rangle F, \Delta}$$

where $x$ and $t$ are of type $s$ and $F$ a first-order formua.

### 9.13.2 Examples

Here is a correct instance of the assignment rule:

$$\frac{z/2 \doteq^{int} y, x \doteq^{s} z + 2 \rightarrow x/2 \doteq^{int} y + 1}{x/2 \doteq^{int} y \rightarrow \langle x = x + 2 \rangle \; x/2 \doteq^{int} y + 1}$$

We may continue the above proof search by

$$\frac{\dfrac{z/2 \doteq^{int} y \rightarrow (z + 2)/2 \doteq^{int} y + 1}{z/2 \doteq^{int} y, x \doteq^{s} z + 2 \rightarrow x/2 \doteq^{int} y + 1}}{x/2 \doteq^{int} y \rightarrow \langle x = x + 2 \rangle \; x/2 \doteq^{int} y + 1}$$

### 9.13.3 Soundness Proof

To prove soundness (see Definition 37 ) of the assignment rule we assume that the premise $\Gamma(z/x), x \doteq t(z/x) \rightarrow F, \Delta(z/x)$ is universally valid and aim to show that the conclusion $\Gamma \rightarrow \langle x = t\rangle F, \Delta$ is also.

We fix a DL-Kripke structure $\mathcal{K} = (S, \rho)$ and an arbitrary state $(\mathcal{A}, \beta)$ in $S$, assuming $(\mathcal{A}, \beta) \models \Gamma$. Let

$$\beta'(u) = \begin{cases} \beta(u) & \text{if} \quad u \neq z \\ \beta(x) & \text{if} \quad u = z \end{cases}$$

Since the variable $z$ is new, it does not occur in $\Gamma$. Therefor we still have $(\mathcal{A}, \beta') \models \Gamma$. But we also have $(\mathcal{A}, \beta') \models \Gamma(z/x)$ by the substitution lemma for first-order formulas. Let $\beta''$ be defined by

$$\beta''(u) = \begin{cases} \beta'(u) & \text{if} \quad u \neq x \\ t^{(\mathcal{A},\beta')} & \text{if} \quad u = x \end{cases}$$

We want to show $(\mathcal{A}, \beta'') \models F$,

First, we note that we still have $(\mathcal{A}, \beta'') \models \Gamma(z/x)$ since $\beta''$ amd $\beta'$ differ only at the variable $x$ which does not occur in $\Gamma(z/x)$. Furthermore we observe $t^{(\mathcal{A},\beta')} = t(z/x)^{(\mathcal{A},\beta')} = t(z/x)^{(\mathcal{A},\beta'')}$. Thus $(\mathcal{A}, \beta'') \models x \doteq t(z/x)$. By assumption we have $(\mathcal{A}, \beta'') \models F, \Delta(z/x)$. If $(\mathcal{A}, \beta'') \models \Delta(z/x)$ is the case, we also have $(\mathcal{A}, \beta') \models \Delta(z/x)$ (since $x$ does not occur in $\Delta(z/x)$) and therefor also $(\mathcal{A}, \beta) \models \Delta$ and we are finished. This leaves us with the case $(\mathcal{A}, \beta'') \models F$ which again finishes the proof, because this shows $(\mathcal{A}, \beta') \models \langle x = t\rangle F$.

Note, that we have used the substitution Lemma 12 with the substitution

$$\sigma(u) = \begin{cases} u & \text{if } u \neq x \\ z & \text{if } u = x \end{cases}$$

It is obvious that $\sigma$ is not an allowed formula for $\Gamma$ or $\Delta$ if $\Gamma$ or $\Delta$ contains none-first-order formulas. In general $x$ will occur again on the left-hand side of an assigment within $\Gamma$ or $\Delta$. Thus there is no chance to generalize the above rule for arbitrary dynamic logic formulas.

■

### 9.13.4 Comments

This rule lies at the very heart of Hoare logic and therefore also of Dynamic logic. It effects the transition from program variables to logic variables. In

the program statement, $x = x + 2$, of the above example the same symbol $x$ appears on both sides of the assignment operator, but different values are associated with these occurrences. No logic variable could do this. The solution is to introduce a new variable, $z$ in this case, which intuitively holds the value of $x$ before execution of the assignment operation. The variable $x$ then holds the value of $x$ after execution of the assignment. In the general rule every occurrence of $x$ intended to refer to the *old* value of $x$ is replaced by $z$. The occurrences of $x$ refering to the *new* value remain unchanged. Notice that these occurrences are the left-hand side of the assignment statement itself and all occurrence of $x$ in the formula $F$. All other occurrences of $x$ refer to the previous value.

**An alternative assigment rule** We again turn the occurences of the variable $x$ in the assignment $x = t$ into different logical variables, but this time we let $x$ denote the value of $x$ before execution and $z$ the value after execution. This yields the rule:

**Definition 38**

$$\frac{\Gamma, z \doteq^s t \rightarrow F(z/x), \Delta}{\Gamma \rightarrow \langle x = t \rangle F, \Delta}$$

*where $x$ and $t$ are of type $s$ and $F$ a first-order formua.*

# 9.14 A Branching Rule

## 9.14.1 The Rule

$$\frac{\Gamma, F_0 \rightarrow \langle \pi_1 \rangle F, \Delta \quad \Gamma, \neg F_0 \rightarrow \langle \pi_2 \rangle F, \Delta}{\Gamma \rightarrow \langle \text{if}(F_0)\{\pi_1\}\,\text{else}\{\pi_2\}\rangle F, \Delta}$$

## 9.14.2 Examples

$$\frac{(\frac{b}{2} * 2 = b) \rightarrow \langle a = 2; \rangle z = y \quad (b/2) * 2 \neq b) \rightarrow \langle z = a \rangle z = y}{\rightarrow \langle \ \text{if}((b/2) * 2 == b)\{a = 2;\}\,\text{else}\{z = a; \rangle z = y}$$

### 9.14.3 Soundness Proof

This is an easy rule. But let us nevertheless spell out the proof. Assuming universal validity of the sequents $\Gamma, F_0 \rightarrow \langle\pi_1\rangle F, \Delta$ and $\Gamma, \neg F_0 \rightarrow \langle\pi_2\rangle F, \Delta$ we set out to prove universal validity of $\Gamma \rightarrow \langle \text{if}(F_0)\{\pi_1\}\,\text{else}\{\pi_2\}\rangle F, \Delta$. Again we fix an arbitrary state $(\mathcal{A}, \beta)$ of an arbitrary Kripke structure $\mathcal{K}$ satisfying $(\mathcal{A}, \beta) \models \Gamma$. There are two cases to be distinguished

1. $(\mathcal{A}, \beta) \models F_0$
2. $(\mathcal{A}, \beta) \models \neg F_0$

Let us follow case (1) here. The other case is absolutely analogous. Our assumption yields $(\mathcal{A}, \beta) \models \langle\pi_1\rangle F, \Delta$. If we have in fact $(\mathcal{A}, \beta) \models \Delta$, then we are through. From now on may assume $(\mathcal{A}, \beta) \models \langle\pi_1\rangle F$

Let $(\mathcal{B}, \gamma)$ be the unique state with $((\mathcal{A}, \beta), (\mathcal{B}, \gamma)) \in \rho(\pi)$ with $\pi \equiv \text{if}(F_0)\{\pi_1\}\,\text{else}\{\pi_2\}$. If we can show $(\mathcal{B}, \gamma) \models F$ we are finished.

Following Definition 28 clause 10 we obtain $((\mathcal{A}, \beta), (\mathcal{B}, \gamma)) \in \rho(\pi_1)$ and using $(\mathcal{A}, \beta) \models \langle\pi_1\rangle F$ we conclude $(\mathcal{B}, \gamma) \models F$.

∎

### 9.14.4 Comments

Since this was so easy let us pause a moment to point out another issue. The rules we have seen so far work both ways. More precisely their conclusion is universally valid *if and only if* all premisses are universally valid, see Exercises 9.18.7 and 9.18.6. What is the significance of this? Let us look at the following first-branch-only rule

$$\frac{\Gamma \rightarrow F_0, \Delta \quad \Gamma \rightarrow \langle\pi_1\rangle F, \Delta}{\Gamma \rightarrow \langle \text{if}(F_0)\{\pi_1\}\,\text{else}\{\pi_2\}\rangle F, \Delta}$$

It can be easily seen that this is a valid rule and there might even be proof situations where it could be usefully employed. But, in general it is a dangerous rule to use: We usually contruct proofs from bottom to top. Our goal is to show that $\Gamma \rightarrow \langle \text{if}(F_0)\{\pi_1\}\,\text{else}\{\pi_2\}\rangle F, \Delta$ is universally valid. If we replace this goal by the two goals $\Gamma \rightarrow F_0, \Delta$ and $\Gamma \rightarrow \langle\pi_1\rangle F, \Delta$ we attempt

to prove something that is stronger than necessary. If we succeed, we are lucky, otherwise we have to backtrack and restart our proof in the situation just before applying the first-branch-only rule. There are more complex situations where we cannot avoid going down blind alleys, but we should try as best we can to avoid this.

## 9.15 A While Rule

### 9.15.1 The Rule

$$\frac{\Gamma \rightarrow I \qquad I, F_0 \rightarrow [\pi]I \qquad I, \neg F_0 \rightarrow F, \Delta}{\Gamma \rightarrow [\text{while}(F_0)\{\pi\}]F, \Delta}$$

### 9.15.2 Example

$$\frac{\begin{array}{c} x \doteq a \wedge y \doteq b \wedge z \doteq 0 \rightarrow a * b + z \doteq x * y \\ a * b + z \doteq x * y, \neg(b \doteq 0) \rightarrow [\alpha_{RMbody}]a * b + z \doteq x * y \\ a * b + z \doteq x * y, b \doteq 0 \rightarrow z \doteq x * y \end{array}}{x \doteq a \wedge y \doteq b \wedge z \doteq 0 \rightarrow [\text{while}(b! = 0)\{\alpha_{RMbody}\}]\ z \doteq x * y}$$

whith $\alpha_{RMbody}$ the example program from Section 9.1. It is only for typographical reasons that we have printed the three premises vertically rather than horizontally.

### 9.15.3 Soundness Proof

We assume the following three sequents to be universally valid

1 $\Gamma \rightarrow I$
2 $I, F_0 \rightarrow [\pi]I$
3 $I, \neg F_0 \rightarrow F, \Delta$

and try to prove $(\mathcal{A}, \beta) \models [\text{while}(F_0)\{\pi\}]F, \Delta$ for all states $(\mathcal{A}, \beta)$ satisfying $(\mathcal{A}, \beta) \models \Gamma$ in all Kripke structure $\mathcal{K}$. Unravelling this proof obligation we need to show for every $n \geq 0$ and every sequence $(\mathcal{A}_i, \beta_i)$, $0 \leq i \leq n$ of states satisfying

$a$ $(\mathcal{A}_0, \beta_0) = (\mathcal{A}, \beta)$
$b$ $((\mathcal{A}_i, \beta_i), (\mathcal{A}_{i+1}, \beta_{i+1})) \in \rho(\pi)$ for all $0 \leq i < n$
$c$ $(\mathcal{A}_i, \beta_i) \models F_0$ for all $0 \leq i < n$
$d$ $(\mathcal{A}_n, \beta_n) \models \neg F_0$

that

$$(\mathcal{A}_n, \beta_n) \models F$$

is true. From assumption (1),(a) and (c) we obtain $(\mathcal{A}_0, \beta_0) \models I \wedge F_0$. Now (2),(b) and (c) imply $(\mathcal{A}_i, \beta_i) \models I \wedge F_0$ for all $0 \leq i < n$. While (2) and (d) imply $(\mathcal{A}_n, \beta_n) \models I \wedge \neg F_0$. Now assumption (3) gives $(\mathcal{A}_n, \beta_n) \models F, \Delta$ as desired.

■

### 9.15.4 Comments

It should be noted that $\Delta$ occurs on the left-hand side only in the conclusion and in the third premise of the `while`-rule. Likewise $\Gamma$ occurs only on the right-hand side in the conclusion and in the first premise. This is crucial for the soundness of the rule, see Exercise 9.18.8.

Since there is a new syntactic entity, $I$, occuring in the premises of the rule, but not in its conclusion, it does not make sense to require the converse direction of the soundness claim.

The given while rule may have its merrits, but it is certainly a disadvantage that termination of the while-loop has to be proved in addition. A possibility to handlethe diamond modality for while-loops will be given int he next Section 9.16

## 9.16 Integer Induction Rule

### 9.16.1 The Rule

$$\frac{\begin{array}{c}\Gamma \rightarrow F(0/z), \Delta \\ \Gamma \rightarrow \forall v : int(v \geq 0 \wedge F(v/z) \rightarrow F((v+1)/z)), \Delta \\ \Gamma, \forall v.int(v \geq 0 \rightarrow F(v/z)) \rightarrow \Delta\end{array}}{\Gamma \rightarrow \Delta}$$

Since there are no requirements placed on $\Gamma$ or $\Delta$ this rule can be always applied. It is a kind of *cut rule*. Apparently the rule sets up a proof by induction on integers. The notation $v : int$ signals that the variable $v$ is of type integer. We refer to the three sequents in the premise respectively as *base case*, *step case* and *use case*. What this has to do with proving `while`-loops will be explained in due course.

### 9.16.2 Soundness Proof

Assume universal validity of

$$\begin{array}{c} \Gamma \rightarrowtail F(0/z), \Delta \\ \Gamma \rightarrowtail \forall v : int(v \geq 0 \wedge F(v/z) \rightarrow F((v+1)/z)), \Delta \\ \Gamma, \forall v : int(v \geq 0 \rightarrow F(v/z)) \rightarrowtail \Delta \end{array}$$

and $(\mathcal{A}, \beta) \models \Gamma$ for an arbitrary state d $(\mathcal{A}, \beta)$ with the aim to prove $(\mathcal{A}, \beta) \models \Delta$. From the base case we get $(\mathcal{A}, \beta) \models F(0/z), Delta$. If $(\mathcal{A}, \beta) \models Delta$ is true we are finished, so we assume $(\mathcal{A}, \beta) \models F(0/z)$. In the same way we obtain from the step case $(\mathcal{A}, \beta) \models v : int(v \geq 0 \wedge F(v/z) \rightarrow F((v+1)/z))$ By the principle of integer induction we get from this $(\mathcal{A}, \beta) \models \forall v : int(v \geq 0 \rightarrow F(v/z))$. For this it is, of course, essential the the interpretation of the sort *int* in the structure $\mathcal{A}$ are the integers. Now the use case yields $(\mathcal{A}, \beta) \models \Delta$ as desired.

∎

### 9.16.3 Examples

The first is a typical example of induction on natural numbers. In the premises we use the appreviations

$$\begin{array}{lll} \Gamma & \text{for} & \{a(0,y) \doteq y, a(s(x),y) = f(a(x,y)), a(f(x),y) = f(a(x,y))\} \\ \Delta & \text{for} & \forall x(a(x,a(x,x)) \doteq a(a(x,x),x)) \end{array}$$

$$\frac{\begin{array}{l} \Gamma \rightarrowtail \forall y,z((a(0,a(y,z)) \doteq a(a(0,y),z))), \Delta \\ \Gamma \rightarrowtail \forall x(\forall y,z(a(x,a(y,z)) \doteq a(a(x,y),z)) \\ \quad \rightarrow \forall y,z(a(s(x),a(y,z)) \doteq a(a(s(x),y),z))), \Delta \\ \Gamma, \forall x \forall y \forall z(a(x,a(y,z)) \doteq a(a(x,y),z) \rightarrowtail \Delta \end{array}}{a(0,y) \doteq y, a(s(x),y) = f(a(x,y)) \rightarrowtail \forall x(a(x,a(x,x)) \doteq a(a(x,x),x))}$$

The second example is less obvious and shows a way to approach total correctness claims for `while`-loops.

Let $\pi$ be the program while$(j > 0)\{r = r + 1; j = j - 1;\}$ and $F$ be the formula $\forall v\langle j = z; r = v; \pi\rangle(j \doteq 0)$.

$$\frac{\begin{array}{c} j \geq 0, r \doteq 0 \rightarrow F(0/z), \langle\pi\rangle(j \doteq 0) \\ j \geq 0, r \doteq 0 \rightarrow \forall v : int(v \geq 0 \wedge F(v/z) \rightarrow F((v+1)/z)), \langle\pi\rangle(j \doteq 0) \\ j \geq 0, r \doteq 0, \forall v.int(v \geq 0 \rightarrow F(v/z)) \rightarrow \langle\pi\rangle(j \doteq 0) \end{array}}{j \geq 0, r \doteq 0 \rightarrow \langle\pi\rangle(j \doteq 0)}$$

It is an easy exercise to show that all thre premises can be shown to be universally valid.

### 9.16.4 Comments

This is by the way an interesting example. An attempt to prove $\forall x(a(x, a(x, x)) \doteq a(a(x, x), x))$ directly by induction on $x$ fails. The generalzation to $\forall x(\forall y, z(a(x, a(y, z)) \doteq a(a(x, y), z))$ is necessary. In the appendix 16.2.1 you will find an input file for this problem for the KeY interactive prover.

## 9.17 Assignments with Side Effects

### 9.17.1 The Rules

$$\frac{\Gamma \rightarrow \langle y = y + 1; x = y; \alpha\rangle F, \Delta}{\Gamma \rightarrow \langle x = ++y; \alpha\rangle F, \Delta}$$

$$\frac{\Gamma \rightarrow \langle x = y; y = y + 1; \alpha\rangle F, \Delta}{\Gamma \rightarrow \langle x = y{+}{+}; \alpha\rangle F, \Delta}$$

### 9.17.2 Soundness

We simply observe that for all pairs of states $((\mathcal{A}, \beta), (\mathcal{B}, \gamma))$ we have

$$((\mathcal{A}, \beta), (\mathcal{B}, \gamma)) \in \rho(x = y{+}{+};) \text{ iff } ((\mathcal{A}, \beta), (\mathcal{B}, \gamma)) \in \rho(x = y; y = y + 1;)$$

Similarly for the post-decrement rule.

### 9.17.3 Comments

As can be seen from the soundness argument the following rules is also sound:

$$\frac{\Gamma, \langle y = y + 1; x = y; \alpha \rangle F \rightarrow \Delta}{\Gamma, \langle x = ++y; \alpha \rangle F \rightarrow \Delta}$$

$$\frac{\Gamma \rightarrow [y = y + 1; x = y; \alpha] F, \Delta}{\Gamma \rightarrow [x = ++y; \alpha] F, \Delta}$$

This shows that for practical purposes it will be useful to write rules in a more general schematic fashion. The above group of rules could e.g. be summarized by the following general rule

> whenever you see on the left- or right-hand side of the sequent arrow $\rightarrow$ within an arbitrary modality, $\langle p \rangle$ or $[p]$ a program $p$ of the form $x = ++y; p_{rem}$, then you are allowed to replace $p$ by $y = y + 1; x = y; p_{rem}$.

We will not go into this level of detail. The interested reader may consult [Beckert *et al.*, 2004] for an example of a rule specification language.

## 9.18 Exercises

**Exercise 9.18.1** *Prove claim 9.9.*

**Exercise 9.18.2** *Which of the following character stings are well-formed Dynamic Logic formulas*

1. $P \rightarrow \langle \texttt{x := t} \rangle\ Q$
2. $P \rightarrow \langle \texttt{t(i++) = u;} \rangle\ Q$
3. $\langle \alpha^* \rangle\ F \leftrightarrow F \vee \langle \alpha \rangle \langle \alpha^* \rangle\ F$
4. $\langle \texttt{x = 1} \rangle x \doteq 1$
5. $[\mathbf{RM}(\mathit{true})]\ \mathit{false}$
6. $(\langle \alpha \rangle\ x \doteq y) \leftrightarrow (\langle \beta \rangle\ x \doteq y)$
7. $\exists x \langle \alpha \rangle \mathit{true}$
8. $(F?; \alpha)^*; \neg F?$

**Exercise 9.18.3** *Show that the following DL formulas are tautologies for any program p.*

1. $\neg\langle p\rangle F \leftrightarrow [p]\neg F)$
2. $\neg[p]F \leftrightarrow \langle p\rangle\neg F)$
3. $[p](F \rightarrow G) \rightarrow (([p]F) \rightarrow [p]G)$

*These are very elementary tautologies, valid in almost any modal logic. The proofs require little more then remembering the definitions.*

**Exercise 9.18.4** *Find a formula $\phi$ in Dynamic Logic (as opposed to first-order logic) such that the claim of Lemma 11 is not true.*

**Exercise 9.18.5** *Provef the soundness of the alternative assignment rule in Definition 38.*

**Exercise 9.18.6** *Prove the inverse of the soundness of the assignment rule from Subsection 9.13.1, i.e. universal validity of the conlcusion of this rule also implies universal validity of its premise.*

**Exercise 9.18.7** *Prove the inverse of the soundness of the branching rule, i.e. universal validity of the conlcusion of this rule also implies universal validity of both of its premises.*

**Exercise 9.18.8** *Consider the following extended version of the while rule*

$$\frac{\Gamma \rightarrow I, \Delta \qquad I, F_0 \rightarrow [\pi]I, \Delta \qquad I, \neg F_0 \rightarrow F, \Delta}{\Gamma \rightarrow [\text{while}(F_0)\{\pi\}]F, \Delta}$$

1. *Give an example to show that this is not a sound rule.*
2. *Could you think of a restriction on $\Delta$ (besides $\Delta = \emptyset$ of course) that would restore the soundness of the extended while rule?*

# Chapter 10

# Set Theory

Many formal specification languages, among them as prime examples **Z** and **B**, use set theoretical concepts and notations, as we have done for the semantics description of UML. This is an appealing choice, because these concepts are easy to understand and accessible without mathematical training. Another advantage is the fact, that there is a well developed mathematical theory of sets. In fact, before set theory was perceived as a foundation for specification languages it was considered as a foundation for all of mathematics. A very intriguing idea: once you accept a few axioms of set theory all mathematical results can be derived from them. In this chapter we will convey a first idea of how this works.

## 10.1 Basics

We will use the Zermelo-Fraenkel (ZF for short) axiom system for set theory. In our presentation we follow the textbook [Takeuti & Zaring, 1971].

The full set of ZF axioms is given in the Appendix 14. The language of ZF set theory is the language for first-order predicate logic with the binary relation symbol $\in$ as its only non-logical symbol. In the formulation of the axioms the equality symbol $=$ is also used. But note, this is introduced in axiom **A1** as an abbreviation for a formula containing only $\in$. More precisely, axiom **A1** states only one implication. The reverse implication, i.e.

$$x = y \rightarrow \forall z(z \in x \leftrightarrow z \in y)$$

has nothing to do with set theory, it is a simple consequence of the congruence axioms

$$x = y \rightarrow (p(z, x) \leftrightarrow p(z, y))$$

for any binary relation symbol $p$.

Any free variables in the axioms are implicitely universally quantified.

Before we go on, we need some notational conventions, otherwise our formulas would soon be unintelligible.

We will use for any formula $\phi(x)$ the syntactical construct $\{x \mid \phi(x)\}$, called a class term. We intuitively think of $\{x \mid \phi(x)\}$ as the collection of all sets $a$ satisfying the formula $\phi(a)$. This is only for notational convenience. The new terms can be eliminated as follows:

| | | |
|---|---|---|
| $y \in \{x \mid \phi(x)\}$ | is replaced by | $\phi(y)$ |
| $\{x \mid \phi(x)\} \in y$ | is replaced by | $\exists u(u \in y \wedge$ |
| | | $\forall z(z \in u \leftrightarrow \phi(z)))$ |
| $\{x \mid \phi(x)\} \in \{y \mid \psi(y)\}$ | is replaced by | $\exists u(\psi(u) \wedge$ |
| | | $\forall z(z \in u \leftrightarrow \phi(z)))$ |

Note, that using a class term $\{x \mid \phi(x)\}$, does by far not imply that $\{x \mid \phi(x)\}$ is a set. For $\phi(x) := x \notin x$ this would immediately result in a contradiction. Only after we can prove that $\exists y(y = \{x \mid \phi(x)\})$ can be derived from the axioms, can we use $\{x \mid \phi(x)\}$ as a set.

Having class terms is already very handy, but still further abbreviations are necessary. Here is the first bunch:

**Definition 39 (Abbreviations for sets)**

| | | | |
|---|---|---|---|
| $\emptyset$ | $=$ | $\{x \mid x \neq x\}$ | |
| $\{a, b\}$ | $=$ | $\{x \mid x = a \vee x = b\}$ | |
| $\{a\}$ | $=$ | $\{a, a\}$ | |
| $\langle a, b\rangle$ | $=$ | $\{\{a\}, \{a, b\}\}$ | *This is called the ordered pair of $a$ and $b$* |

Note, that some of these abbreviations have already been used in the axioms in Appendix 14.

Let us look at some easy logical derivations from the ZF axioms.

**Lemma 16** *The following formulas follow from the ZF axioms*

1. $\exists x(x = \emptyset)$
2. $\forall x, y \exists z(z = \{x, y\})$
3. $\forall x \exists z(z = \{x\})$
4. $\forall x, y \exists z(z = \langle x, y\rangle)$

**Proof:** The first step in all four proofs will be to unfold the abbreviating notation of class terms.

1. In a first step we eliminate the $=$ symbol in $\exists x(x = \emptyset)$ using the extensionality axiom, which yields: $\exists x \forall u(u \in x \leftrightarrow u \in \emptyset)$. Now the class term $\emptyset$ is replaced as explained above: $\exists x \forall u(u \in x \leftrightarrow u \neq u)$. Since $u \neq u$ is contradictory, this is equivalent to $\exists x \forall u(u \in x \rightarrow u \neq u)$. Which is logically equivalent to $\exists x \forall u(u \notin x)$, and this is Axiom **A4**.

2. Eliminating $=$ and the class term in $\forall x, y \exists z(z = \{x, y\})$ yields $\forall x, y \exists z \forall u(u \in z \leftrightarrow u = x \vee u = y)$. This is, after renaming of variables, axiom **A5**.

3. Special case of 2.

4. Unfolding the definition of an ordered pair, we get $\forall x, y \exists z(z = \{\{x\}, \{x, y\}\})$.
   In first-order logic it is possible to replace universal quantification in a theorem $\forall w \psi$ of a theory $T$ by $\forall \vec{w} \psi[t/w]$, i.e. replace all occurences of $w$ by the term $t$ and change the universal quantifier $\forall w$ to $\forall \vec{w}$, where $\vec{w}$ are all variables in $t$. Can this also be done with class terms? In general the answer is, no. But, if we can prove for a class term $ct$ $\forall \vec{w} \exists u(u = ct)$, then the same replacement principle is true in ZF. Now, claim 4 follows from 2 and 3.

∎

**Lemma 17**

1. *If $a$ and $b$ are sets, then there is a set $c$ satisfying*

$$\forall z(z \in c \leftrightarrow z \in a \wedge z \in b)$$

   *$c$ is called the intersection of $a$ and $b$, in symbols $c = a \cap b$.*

2. *If $a$ and $b$ are sets, then there is a set $c$ satisfying*

$$\forall z(z \in c \leftrightarrow z \in a \vee z \in b)$$

   *$c$ is called the union of $a$ and $b$, in symbols $c = a \cup b$.*

3. *If $A$ is a non-empty class term, then there is a set $c$ satisfying*

$$\forall z(z \in c \leftrightarrow \forall u(u \in A \rightarrow z \in u))$$

*$c$ is called the intersection of $A$, in symbols $c = \bigcap A$.*

4. *If $a$ is a set, then there is a set $c$ satisfying*

$$\forall z(z \in c \leftrightarrow \exists u(u \in a \wedge z \in u))$$

*$c$ is called the union of $a$, in symbols $c = \bigcup a$.*

**Proof:** Let us for the moment be pedantic.

1. This requires the subset axiom A3

$$\exists y \forall z(z \in y \leftrightarrow z \in x \wedge \phi(z)).$$

We replace the free variable $x$ by $a$, the formula $\phi(z)$ by $z \in b$ and name the element whose existence is guaranteed by the axiom $c$. This leads to

$$\forall z(z \in c \leftrightarrow z \in a \wedge z \in b)$$

as required.

2. Despite the fact that set theoretical union is such a simple concept, it does need two axioms to guarantee its existence. From the pair axioms, A5, we get the existence of a the set $d = \{a, b\}$ and the sum axiom, A7 yields the existence of a set $c$ satisfying

$$\forall z(z \in c \leftrightarrow \exists u(u \in d \wedge z \in u))$$

Substituting $d = \{a, b\}$ yields the claim.

3. Let $A = \{u \mid \psi(u)\}$. Since $A$ is assumed to be non-empty, we may pick an arbitrary element $b \in A$, i.e. an arbitrary $b$ such that $\psi(b)$ is true. Let $\phi(z)$ be the formula $\forall u(\psi(u) \rightarrow z \in u)$. We will, again, use Axiom A3

$$\exists y \forall z(z \in y \leftrightarrow z \in b \wedge \phi(z)).$$

The element, whose existence is guaranteed is named $c$. This yields the claim, when we observe the trivial equivalence

$$\forall z \forall u(\psi(u) \rightarrow z \in u) \leftrightarrow z \in b \wedge \forall u(\psi(u) \rightarrow z \in u)$$

4. Use axiom A7.

∎

Likewise it is easy to prove

**Lemma 18**
$\forall x_1, x_2, y_1, y_2( \quad \langle x_1, x_2 \rangle = \langle y_1, y_2 \rangle \leftrightarrow x_1 = y_1 \wedge x_2 = y_2 \quad )$

**Definition 40**

1. *A relation $r$ is a set of ordered pairs, i.e.*
   $rel(r) \equiv \forall x(x \in r \rightarrow \exists x_1, x_2(x = \langle x_1, x_2 \rangle))$
2. *The relation $r$ is said to be a relation on the set $s$ if*
   $rel(r, s) \equiv rel(r) \wedge \forall x_1, x_2(\langle x_1, x_2 \rangle \in r \rightarrow x_1 \in s \wedge x_2 \in s)$
3. *A function is a* one-valued *relation, i.e.*
   $func(r) \equiv rel(r) \wedge \forall x, y_1, y_2(\langle x, y_1 \rangle \in r \wedge \langle x, y_2 \rangle \in r \rightarrow y_1 = y_2)$
4. *A function $f$ is said to be a function from a set $a$ to a set $b$ if*
   $func(f, a, b) \equiv func(f) \wedge \forall x_1, x_2(\langle x_1, x_2 \rangle \in f \rightarrow x_1 \in a \wedge x_2 \in b)$

**Lemma 19**
*From the ZF axioms we can prove for any sets $a$, $b$ the existence of the set of all relations on $a$ and of all functions from $a$ to $b$, i.e.*

1. $\forall x \exists y \forall z(z \in y \leftrightarrow rel(z, x))$
2. $\forall u, w \exists y \forall z(z \in y \leftrightarrow func(z, u, w))$

**Proof:** For this proof we need (for the first time in this text) the power set axiom $\exists y \forall z(z \in y \leftrightarrow \forall u(u \in z \rightarrow u \in x))$. We denote the set whose existence is stipulated by this axiom by $\mathcal{P}(x)$.

1. For any set $a$ the set $\mathcal{P}(\mathcal{P}(x))$ exists. The set $c$ of all relations on $a$ is a subset of this set. Since we can describe by a first-order formula exactly which elements of $\mathcal{P}(\mathcal{P}(x))$ belong to $c$ we get the existence of $c$ by the subset axiom.
2. Similar.

∎

## 10.2 The Natural Numbers

**Definition 41 (Successor)** *For any set $a$ the set*

$$a^+ = a \cup \{a\}$$

*is called the successor set of $a$.*

From our previous results it is obvious that $a^+$ is a set, when $a$ is. In the following we will no longer mention facts of this simple kind explicitely.

We will use the empty set $\emptyset$ to represent the natural number 0, $\emptyset^+ = \{\emptyset\} = \{0\}$ to represent 1, $1^+ = \emptyset^{++} = \{\emptyset, \{\emptyset\}\} = \{0, 1\}$ to represent 2. In general, for any natural number $n$ we let $n^+$ represent its successor. We want the set of natural numbers to be $\mathbb{N} = \{0, 0^+, 0^{++}, 0^{+++}, \ldots\}$. It remains to be explain how this can be turned into a legal definition and prove the existence of $\mathbb{N}$ from the ZF axioms.

**Definition 42** *A set $a$ is called a Dedekind set if $0 \in a$ and for all $b \in a$ also $b^+ \in a$. In symbols $Ded(a) \equiv 0 \in a \wedge \forall x(x \in a \rightarrow x^+ \in a)$.*

**Lemma 20**

$$\exists y(y = \bigcap\{a \mid Ded(a)\}$$

*can be derived from the ZF axioms.*

*$\bigcap\{a \mid Ded(a)\}$ will be called the set of natural numbers and denoted by $\mathbb{N}$. In set theory it is also customary to use the symbol $\omega$ instead of $\mathbb{N}$.*

**Proof:** The claim follows from Lemma 17(3) if we can show that there is at least on set $a$ with $Ded(a)$. But this is guaranteed by Axiom A8, the infinity axiom.

■

The Peano axiom system is usually taken as an axiomatic characterisation of the natural numbers. In the context of set theory we should be able to derive them from the set theoretical axioms.

**Lemma 21**
*The following theorems can be derived from the ZF axioms*

1. $0 \in \mathbb{N}$.
2. *If* $n \in \mathbb{N}$ *then* $n^+ \in \mathbb{N}$.
3. $\forall n(n \in \mathbb{N} \rightarrow n^+ \neq 0)$.
4. $\forall n, m(n \in \mathbb{N} \wedge m \in \mathbb{N} \wedge n^+ = m^+ \rightarrow n = m)$.
5. $\forall x(0 \in x \wedge \forall y(y \in x \rightarrow y^+ \in x) \rightarrow \mathbb{N} \subseteq x)$.

**Proof:** 1 and 2 are obvious by definition of $\mathbb{N}$.

To prove 3 we note that $n \in n^+$ is true for any $n$, thus $n^+$ cannot be the empty set.

Assume for a proof of 4 that $n^+ = m^+$, i.e. $n \cup \{n\} = m \cup \{m\}$. Thus we must have

1. $m \in n \cup \{n\}$, i.e. $n = m$ or $m \in n$.
2. $n \in m \cup \{m\}$, i.e. $n = m$ or $n \in m$.

The foundation axiom, A2,

$$\exists y(y \in x) \rightarrow \exists y(y \in x \wedge \forall z \neg(z \in x \wedge z \in y)),$$

instantiated for $x = \{n, m\}$ yields after some simplifications $n \notin m$ or $m \notin n$. Thus the above case distinction forces $n = m$.

Part 5 is again simple. Any $x$ satisfying the premise of the implication is a Dedekind set. Since $\mathbb{N}$ is by definition the intersection of all Dedekind set,we obviously get $\mathbb{N} \subseteq x$.

∎

## 10.3 Comments

1. ZF is by far the most common axiom system for set theory. Others are the Neumann-Bernays-Gödel system (this is e.g. used in [Rubin, 1967]) and the Taski-Grothendiek system.

2. Notice, that ZF set theory is a theory of first-order logic, despite the fact that sets are involved, which are usually thought of as second-order objects. The point here is, that the classification into second-order, third-order and so on is relative to a fixed level of first-order elements. In set theory sets are first-order elements.

3. There are versions of set theory that start out with an initial set of elements of arbitrary kind, usually called *urelements*. On top of these set theory is built, i.e. there will be set of urelements, sets of sets of urelements and so on. In our exposition we are interested in reduction to first principles, so it makes sense to go all the way and consider nothing but sets.

4. We hae chosen the textbook [Takeuti & Zaring, 1971]as a reference mainly for the reason that it is explicitely mention in the ANSI standard draft for **Z**. A very gentle, but rigorous introduction may be found in [Halmos, 1994, Halmos, 1974].

## 10.4 Exercises

**Exercise 10.4.1** *The element-of-relation $\in$ is not transitive, i.e. for most sets $a$, $b$, $c$ the premises $a \in b$ and $b \in c$ do* **not** *imply $a \in c$. There are, however, special circumstance under which this implication is true.*

***Definition 43*** *A set $a$ is called* transitive *if for all $b \in a$ also $b \subset a$ is true.*

*Prove the following*

1. *If $a \in b$ and $b \in c$ and $c$ is a transitive set, then $a \in c$.*

2. *If $a$ is a transitive set, then $a^+$ is also.*

**Exercise 10.4.2**

1. *Let $<$ be the usual order on $\mathbb{N}$. Show for all $a, b \in \mathbb{N}$:*

$$a < b \Leftrightarrow a \in b$$

2. *For any $a \in \mathbb{N}$* $\quad a = \{b \mid b < a\}$

**Exercise 10.4.3**

***Definition 44*** *A set $a$ is called an* ordinal *if it is transitive and satisfies* $\forall x \forall y (x \in a \wedge y \in a \rightarrow (x \in y \vee y \in x \vee x = y)$

*Show: If $a$ is an ordinal then $a^+$ is also.*

# Chapter 11

# Solutions to Exercises

## 11.1 Solutions to Chapter 2

## 11.2 Solutions to Chapter 3

## 11.3 Solutions to Chapter 4

**Exercise 4.13.1**
By Definition we have the direct subtype relations $Integer < Real$, $Set(Integer) < Set(Real)$ and $Set(Real) < Collection(Real)$. Thus $Set(Integer) \ll Collection(Real)$.

**Exercise 4.13.2**

1. **context** *s*:Supplier
   **inv** s.delivery_days.good = s.order

2. **context** *s*:Supplier
   **inv** s.delivery_days -> forAll(d:Date | d.good = s.order)

## 11.4 Solutions to Chapter 9

**Exercise 9.18.1**
It suffices to show that for all integers $a, b, z$

$$a * b + z = (2 * a) * (b/2) + z + a$$

By assumption $b$ is an odd number, i.e. $b = 2 * b_0 + 1$. Then $(b/2) = b_0$ and the equation we want to prove can be rewritten as

$$a * b + z = (2 * a) * b_0 + z + a$$

or

$$a * b + z = a * (2 * b_0 + 1) + z$$

Which is immediatedly seen to be correct.

**Exercise 9.18.3**
Let $\mathcal{K} = (S, \rho)$ be an arbitrary Kripke structure and $s \in S$ an arbitrary state.

1. $(\mathcal{K}, s) \models \langle p \rangle F$ is true if there is a state $s' \in S$ such that $(s, s') \in \rho(p)$ and $(\mathcal{K}, s') \models F$. Negating this statement we see that $(\mathcal{K}, s) \models \neg\langle p \rangle F$ is true if for all states $s' \in S$ satisfying $(s, s') \in \rho(p)$ we get $(\mathcal{K}, s') \models \neg F$. But this is exactly the definition of $(\mathcal{K}, s) \models [p]\neg F$

2. Completely analogous to 1.

3. This is the well-known axiom **K** for normal modal logics.

   Assume that $(\mathcal{K}, s) \models [p](F \to G)$ and $(\mathcal{K}, s) \models [p]F$ with the aim to show $(\mathcal{K}, s) \models [p]G$. By the assumptions we get for any $s' \in S$ with $(s, s') \in \rho(p)$ both $(\mathcal{K}, s') \models F \to G$ and $(\mathcal{K}, s') \models F$. Thus we conclude $(\mathcal{K}, s') \models G$ for any such $s'$, i.e. $(\mathcal{K}, s) \models [p]G$, indeed.

**Exercise 9.18.4**
Consider $\phi \equiv \langle x = 5 \rangle (1 = (\text{if } x = y \text{ then } 0 \text{ else } 1))$. By definition $\phi_1 \equiv \langle x = 5 \rangle (1 = 0)$ and $\phi_2 \equiv \langle x = 5 \rangle (1 = 1)$.

Let $(\mathcal{A}, \beta)$ be a state with $\beta(x) = \beta(y) = 0$. Then $(\mathcal{A}, \beta) \models \phi$ and by Lemma 11 we should have $(\mathcal{A}, \beta) \models \phi \leftrightarrow \phi_1$. But certainly $(\mathcal{A}, \beta) \models \neg\langle x = 5 \rangle (1 = 0)$.

**Exercise 9.18.5**
We start with the assumption that $\Gamma \rightarrow \langle x = t \rangle F, \Delta$ is universally valid. To show that the conclusion of the rule is also valid we fix an arbitrary state $(\mathcal{A}, \beta)$ with $(\mathcal{A}, \beta) \models \Gamma \wedge z \doteq^s t$. For later use we note that this entails in particular

$$\beta(z) = t^{(\mathcal{A}, \beta)}$$

Since we assued the premisse of the rule to be universally valid we must also have $(\mathcal{A}, \beta) \models \langle x = t \rangle F \vee \Delta$. If we had in fact $(\mathcal{A}, \beta) \models \Delta$ then we are through. We thus assume from here on $(\mathcal{A}, \beta) \models \langle x = t \rangle F$. By the semantics of the assignment statement this yields $(\mathcal{A}, \beta') \models F$ with

$$\beta'(u) = \begin{cases} \beta(u) & \text{if } u \neq x \\ t^{(\mathcal{A}, \beta)} & \text{if } u = x \end{cases}$$

Remember, we aim to show $(\mathcal{A}, \beta) \models F(z/x)$, which by Lemma 12 is equivalent to $(\mathcal{A}, \beta'') \models F$ with

$$\beta''(u) = \begin{cases} \beta(u) & \text{if } u \neq x \\ \beta(z) & \text{if } u = x \end{cases}$$

If we can thus show $\beta' = \beta''$ we are done. Here we go!

$$\begin{array}{ll} \text{for } u \neq x & \beta'(u) = \beta(u) = \beta''(u) \\ \text{for } u = x & \beta'(u) = t^{(\mathcal{A},\beta)} = \beta(z) = \beta''(x) \end{array}$$

■

We note again that this proof cannot be extended to general formulas $F$. The substitution $\sigma$ that was used in the application of the substitution lemma in the above proof:

$$\sigma(u) = \begin{cases} u & \text{if } u \neq x \\ z & \text{if } u = x \end{cases}$$

will in general not be allowed for arbitrary $F$.

**Exercise 9.18.6**

We assume that the conclusion $\Gamma \rightarrow \langle x = t\rangle F, \Delta$ is universally valid and aim to prove that the premise $\Gamma(z/x), x \doteq^s t(z/x) \rightarrow F, \Delta(z/x)$ is also. We fix a DL-Kripke structure $\mathcal{K} = (S, \rho)$ and an arbitrary state $(\mathcal{A}, \beta)$ in $S$, assuming $(\mathcal{A}, \beta) \models \Gamma(z/x), x \doteq^s t(z/x)$. We have to show $(\mathcal{A}, \beta) \models F, \Delta(z/x)$. Let

$$\beta\prime(u) = \begin{cases} \beta(u) & \text{if} \quad u \neq x \\ \beta(z) & \text{if} \quad u = x \end{cases}$$

Then $(\mathcal{A}, \beta) \models \Gamma(z/x)$ implies $(\mathcal{A}, \beta') \models \Gamma$. Since by assumption $\Gamma \rightarrow \langle x = t\rangle F, \Delta$ is universally valid, we get $(\mathcal{A}, \beta') \models \langle x = t\rangle F, \Delta$. In case, $(\mathcal{A}, \beta') \models \Delta$ is true, we immediately get $(\mathcal{A}, \beta) \models \Delta(z/x)$ and thus also $(\mathcal{A}, \beta) \models F, \Delta(z/x)$. The remaining case, $(\mathcal{A}, \beta') \models \langle x = t\rangle F$ is of course the crucial one. By Definition 28 this implies $(\mathcal{A}, \beta'') \models F$ with

$$\beta''(u) = \begin{cases} \beta(u) & \text{if} \quad u \neq x \\ t^{(\mathcal{A},\beta')} & \text{if} \quad u = x \end{cases}$$

We observe that $\beta''(u) = \beta'(u) = \beta(u)$ for $u \neq x$ and $\beta''(x) = t^{(\mathcal{A},\beta')} = t(z/x)^{(\mathcal{A},\beta)}$. Since by assumption $(\mathcal{A}, \beta) \models x \doteq^s t(z/x)$, we get $\beta''(x) = t(z/x)^{(\mathcal{A},\beta)} = \beta(x)$. Thus $(\mathcal{A}, \beta'') \models F$ implies $(\mathcal{A}, \beta) \models F$ and we are done.

■

**Exercise 9.18.7**
We assume that $\Gamma \rightarrow \langle \text{if}(F_0)\{\pi_1\}\,\text{else}\{\pi_2\}\rangle F, \Delta$ is universally valid and fix a state $(\mathcal{A}, \beta)$ in an arbitrary Kripke structure $\mathcal{K}$ with the aim to show $(\mathcal{A}, \beta) \models \Gamma, F_0 \rightarrow \langle \pi_1 \rangle F, \Delta$ and $(\mathcal{A}, \beta) \models \Gamma, \neg F_0 \rightarrow \langle \pi_2 \rangle F, \Delta$. We are done if $(\mathcal{A}, \beta) \models \neg \bigwedge \Gamma$. So we assume from now on $(\mathcal{A}, \beta) \models \bigwedge \Gamma$. There are naturally two cases to be distinguished:

1. $(\mathcal{A}, \beta) \models F_0$
2. $(\mathcal{A}, \beta) \models \neg F_0$

We only treat the first case. The second is completely analogous. From the assumption there is a state $(\mathcal{B}, \gamma)$ satisfying $((\mathcal{A}, \beta), (\mathcal{B}, \gamma)) \in \rho(\pi)$ and $(\mathcal{B}, \gamma) \models F$. Here $\pi$ stands for the program $\text{if}(F_0)\{\pi_1\}\,\text{else}\{\pi_2\}$. Definition 28 implies under the given circumstances $((\mathcal{A}, \beta), (\mathcal{B}, \gamma)) \in \rho(\pi_0)$ and thus we have indeed proved $(\mathcal{A}, \beta) \models \Gamma, F_0 \rightarrow \langle \pi_1 \rangle F, \Delta$.

∎

**Exercise 9.18.8**

1. Let us instantiate the extended while rule with $\Gamma = \mathit{empty}$, $I = \mathit{true}$, $F \equiv F_0 \equiv x \neq 1$, $\Delta = \{x = 1\}$

$$\frac{\emptyset \rightarrow \mathit{true} \qquad x \neq 1 \rightarrow [x = 1]\,\mathit{true} \qquad x = 1 \rightarrow x \neq 1, x = 1}{\emptyset \rightarrow [\text{while } (x \neq 1)\{x = 1\}]x \neq 1, x = 1}$$

   The bottom sequent is not universally valid, but all sequents in the conclusion of the rule (assuming that we apply rules from bottom to top) are. This exampe has been supplied by Philipp Rümmer.

2. I helps to assume that for any pair os states $((\mathcal{A}, \beta), (\mathcal{B}, \gamma)) \in \rho(\pi)$ with $(\mathcal{A}, \beta) \models \Delta$ we also have $(\mathcal{B}, \gamma) \models \Delta$. In other words, we assume that the program $\pi$ only changes parts of the vocabulary that is not relevant for $\Delta$.

## 11.5 Solutions to Chapter 10

**Exercise 10.4.1**

1. Since $c$ is transitive, $b \in c$ implies $b \subset c$. Now, we get from $a \in b$ immediately $a \in c$.

2. Assuming $a$ to be transitive we want to prove that $a \cup \{a\}$ is transitive. Consider thus $b \in a \cup \{a\}$. In case $a = b$, we get immediately $b = a \subset a \cup \{a\}$. The only other possibility is $b \in a$. By assumption this yields $b \subset a$ and thus also $b \subset a \cup \{a\}$.

**Exercise 10.4.2**

1. The usual order relation $<$ on $\mathbb{N}$ is uniquely determined by the requirements $n < n^+$ for all $n \in \mathbb{N}$ and $<$ is a transitive relation on $\mathbb{N}$. We only need to convince ourselves that $\in$ has these two properties. The first is obvious by the definition of $n^+$. The second property follows from the previous Exercise 10.4.1.

2. Trivial implication of 1.

**Exercise 10.4.3**

# Chapter 12

# Appendix: Predefined OCL Types

This section contains all standard types defined within OCL, including all the properties defined on those types as listed in UML V1.3 June 1999. Its signature and a description of its semantics define each property. Within the description, the reserved word result is used to refer to the value that results from evaluating the property. In several places, post conditions are used to describe properties of the result. When there is more than one postcondition, all postconditions must be true.

## 12.1 Basic Types

The basic types used are Integer, Real, String, and Boolean.

### 12.1.1 Integer

The OCL type Integer represents the mathematical concept of integer.

Properties of Integer, where the instance of Integer is called i.

**i = (i2 : Integer) : Boolean** True if i is equal to i2.

**i + (i2 : Integer) : Integer** The value of the addition of i and i2.

**i - (i2 : Integer) : Integer** The value of the subtraction of i2 from i.

**i * (i2 : Integer) : Integer** The value of the multiplication of i and i2.

**i / (i2 : Integer) : Real** The value of i divided by i2.

**i.abs : Integer** The absolute value of i.
**post**: if i < 0 then result = - i else result = i endif

**i.div( i2 : Integer) : Integer** The number of times that i2 fits completely within i.
**pre** : i2 <> 0
**post**: if i / i2 >= 0 then result = (i / i2).floor else result = -((-i/i2).floor) endif

**i.mod( i2 : Integer) : Integer** The result is i modulo i2.
**post**: result = i - (i.div(i2) * i2)

**i.max(i2 : Integer) : Integer** The maximum of i an i2.
**post**: if i >= i2 then result = i else result = i2 endif

**i.min(i2 : Integer) : Integer** The minimum of i an i2.
**post**: if i <= i2 then result = i else result = i2 endif

### 12.1.2 Real

The OCL type Real represents the mathematical concept of real. Note that Integer is a subclass of Real, so for each parameter of type Real, you can use an integer as the actual parameter.

Properties of Real, where the instance of Real is called r.

**r = (r2 : Real) : Boolean** True if r is equal to r2.

**r <> (r2 : Real) : Boolean** True if r is not equal to r2.
**post**: result = not (r = r2)

**r + (r2 : Real) : Real** The value of the addition of r and r2.

**r - (r2 : Real) : Real** The value of the subtraction of r2 from r.

**r * (r2 : Real) : Real** The value of the multiplication of r and r2.

**r / (r2 : Real) : Real** The value of r divided by r2.

**r.abs : Real** The absolute value of r.
**post**: if r < 0 then result = - r else result = r endif

**r.floor : Integer** The largest integer which is less than or equal to r.
**post**: (result <= r) and (result + 1 > r)

**r.round : Integer** The integer which is closest to r. When there are two such integers, the largest one.
**post**: ((r - result) < r).abs < 0.5) or ((r - result).abs = 0.5 and (result > r))
`hier stimmt doch was nicht`

**r.max(r2 : Real) : Real** The maximum of r and r2.
**post**: if r >= r2 then result = r else result = r2 endif

**r.min(r2 : Real) : Real** The minimum of r and r2.
**post**: if r <= r2 then result = r else result = r2 endif

**r < (r2 : Real) : Boolean** True if r1 is less than r2.

**r > (r2 : Real) : Boolean** True if r1 is greater than r2.
**post**: result = not (r <= r2)

**r <= (r2 : Real) : Boolean** True if r1 is less than or equal to r2.
**post**: result = (r = r2) or (r < r2)

**r >= (r2 : Real) : Boolean** True if r1 is greater than or equal to r2.
**post**: result = (r = r2) or (r > r2)

### 12.1.3 String

The OCL type String represents ASCII strings.

Properties of String, where the instance of String is called string.

**string = (string2 : String) : Boolean** True if string and string2 contain the same characters, in the same order.

**string.size : Integer** The number of characters in string.

**string.concat(string2 : String) : String**
The concatenation of string and string2.
**post**: result.size = string.size + string2.size
**post**: result.substring(1, string.size) = string
**post**: result.substring(string.size + 1, result.size) = string2

**string.toUpper : String** The value of string with all lowercase characters converted to uppercase characters.
**post**: result.size = string.size

**string.toLower : String** The value of string with all uppercase characters converted to lowercase characters.
**post**: result.size = string.size

**string.substring(lower : Integer, upper : Integer) : String** The sub-string of string starting at character number lower, up to and including character number upper.

### 12.1.4 Boolean

The OCL type Boolean represents the common true/false values.

Features of Boolean, the instance of Boolean is called b.

**b = (b2 : Boolean) : Boolean** Equal if b is the same as b2.

**b or (b2 : Boolean) : Boolean** True if either b or b2 is true.

**b xor (b2 : Boolean) : Boolean** True if either b or b2 is true, but not both.
**post**: (b or b2) and not (b = b2)

**b and (b2 : Boolean) : Boolean** True if both b1 and b2 are true.

**not b : Boolean** True if b is false.
**post**: if b then result = false else result = true endif

**b implies (b2 : Boolean) : Boolean** True if b is false, or if b is true and b2 is true.
**post**: (not b) or (b and b2)

**if b then (expression1 : OclExpression) else (expression2: OclExpression) endif : expression1.evaluationType** If b is true, the result is the value of evaluating expression1; otherwise, result is the value of evaluating expression2.

## 12.2 Enumeration

The OCL type Enumeration represents the enumerations defined in an UML model.

Features of Enumeration, the instance of Enumeration is called enumeration.

**enumeration = (enumeration2 : Boolean) : Boolean**
Equal if enumeration is the same as enumeration2.

**enumeration <> (enumeration2 : Boolean) : Boolean** Equal if enumeration is not the same as enumeration2.
**post**: result = not ( enumeration = enumeration2)

## 12.3 Collection-Related Types

The following sections define the properties on collections (i.e., these properties are available on Set, Bag, and Sequence). As defined in this section, each collection type is actually a template with one parameter. T denotes the parameter. A real collection type is created by substituting a type for the T. So Set (Integer) and Bag (Person) are collection types.

### 12.3.1 Collection

Collection is the abstract supertype of all collection types in OCL. Each occurrence of an object in a collection is called an element. If an object occurs twice in a collection, there are two elements. This section defines the properties on Collections that have identical semantics for all collection subtypes. Some properties may be defined with the subtype as well, which means that there is an additional postcondition or a more specialized return value. The definition of several common properties is different for each subtype. These properties are not mentioned in this section.

Properties of Collection, where the instance of Collection is called collection.

**collection->size : Integer** The number of elements in the collection collection.
**post**: result = collection->iterate(elem; acc : Integer = 0 | acc + 1)

**collection->includes(object : OclAny) : Boolean** True if object is an element of collection, false otherwise.
**post**: result = (collection->count(object) > 0)

**collection->excludes(object : OclAny) : Boolean** True if object is not an element of collection, false otherwise.
**post**: result = (collection->count(object) = 0)

**collection->count(object : OclAny) : Integer**
The number of times that object occurs in the collection collection.
**post**: result = collection->iterate( elem; acc : Integer = 0 | if elem = object then acc + 1 else acc endif)

**collection->includesAll(c2 : Collection(T)) : Boolean**
Does collection contain all the elements of c2 ?
**post**: result = c2->forAll(elem | collection->includes(elem))

**collection->excludesAll(c2 : Collection(T)) : Boolean**
Does collection contain none of the elements of c2 ?
**post**: result = c2->forAll(elem | collection->excludes(elem))

**collection->isEmpty : Boolean** Is collection the empty collection?
**post**: result = ( collection->size = 0 )

**collection->notEmpty : Boolean** Is collection not the empty collection?
**post**: result = ( collection->size <> 0 )

**collection->sum : T** The addition of all elements in collection. Elements must be of a type supporting the + operation. The + operation must take one parameter of type T and be both associative: (a+b)+c = a+(b+c), and commutative: a+b = b+a. Integer and Real fulfill this condition.
**post**: result = collection->iterate( elem; acc : T = 0 | acc + elem )

**collection->exists(expr : OclExpression) : Boolean** Results in true if expr evaluates to true for at least one element in collection.
**post**: result = collection->iterate(elem; acc : Boolean = false | acc or expr)

**collection->forAll(expr : OclExpression) : Boolean** Results in true if expr evaluates to true for each element in collection; otherwise, result is false.
**post**: result = collection->iterate(elem; acc : Boolean = true | acc and expr)

**collection->isUnique(expr : OclExpression) : Boolean**
Results in true if expr evaluates to a different value for each element in collection; otherwise, result is false.
**post**: result = collection->collect(expr)->forAll(e1, e2 | e1 <> e2)

**collection->sortedBy(expr : OclExpression) : Boolean**
Results in the Sequence containing all elements of collection. The element for which expr has the lowest value comes first, and so on. The

type of the expr expression must have the < operation defined. The operation < must be transitive i.e. if a < b and b < c then a < c.
**post**: collection->iterate(expr : OclExpression):expr.evaluationType Iterates over the collection. This is the basic collection operation with which the other collection operations can be described.

### 12.3.2 Set

The Set is the mathematical set. It contains elements without duplicates.

Features of Set, the instance of Set is called set.

**set->union(set2 : Set(T)) : Set(T)** The union of set and set2.
**post**: result->forAll(elem | set->includes(elem) or set2->includes(elem))
**post**: set->forAll(elem | result->includes(elem))
**post**: set2->forAll(elem | result->includes(elem))

**set->union(bag : Bag(T)) : Bag(T)** The union of set and bag.
**post**: result->forAll(elem | result->count(elem) = set->count(elem) + bag->count(elem))
**post**: set->forAll(elem | result->includes(elem))
**post**: bag->forAll(elem | result->includes(elem))

**set = (set2 : Set(T)) : Boolean** Evaluates to true if set and set2 contain the same elements.
**post**: result = (set->forAll(elem | set2->includes(elem))
and set2->forAll(elem | set->includes(elem)) )

**set->intersection(set2 : Set(T)) : Set(T)** The intersection of set and set2 (i.e, the set of all elements that are in both set and set2).
**post**: result->forAll(elem | set->includes(elem)
and set2->includes(elem))
**post**: set->forAll(elem | set2->includes(elem) =
result->includes(elem))
**post**: set2->forAll(elem | set->includes(elem) =
result->includes(elem))

**set->intersection(bag : Bag(T)) : Set(T)** The intersection of set and bag.
**post**: result = set->intersection( bag->asSet )

**set->(set2 : Set(T)) : Set(T)** The elements of set, which are not in set2.
**post**: result->forAll(elem | set->includes(elem)
and set2->excludes(elem))
**post**: set->forAll(elem | result->includes(elem) =
set2->excludes(elem))

**set->including(object : T) : Set(T)** The set containing all elements of set plus object.
**post**: result->forAll(elem | set->includes(elem) or (elem = object))
**post**: set->forAll(elem | result->includes(elem))
**post**: result->includes(object)

**set->excluding(object : T) : Set(T)** The set containing all elements of set without object.
**post**: result->forAll(elem | set->includes(elem) and (elem <> object))
**post**: set->forAll(elem | result->includes(elem) = (object <> elem))
**post**: result->excludes(object)

**set->symmetricDifference(set2 : Set(T)) : Set(T)** The set containing all the elements that are in set or set2, but not in both.
**post**: result->forAll(elem | set->includes(elem) xor
set2->includes(elem))
**post**: set->forAll(elem | result->includes(elem) =
set2->excludes(elem))
**post**: set2->forAll(elem | result->includes(elem) =
set->excludes(elem))

**set->select(expr : OclExpression): Set(T)** The subset of set for which expr is true.
**post**: result = set->iterate(elem; acc : Set(T) = Set | if expr then acc->including(elem) else acc endif)

**set->reject(expr : OclExpression): Set(T)** The subset of set for which expr is false.
**post**: result = set->select(not expr)

**set->collect(expr : OclExpression) : Bag(expr.evaluationType)**
The Bag of elements which results from applying expr to every member of set.
**post**: result = set->iterate(elem; acc : Bag(expr.evaluationType) = Bag | acc->including(expr) )

**set->count(object : T) : Integer** The number of occurrences of object in set.
**post**: result ¡= 1

**set->asSequence : Sequence(T)** A Sequence that contains all the elements from set, in undefined order.
**post**: result->forAll(elem | set->includes(elem))
**post**: set->forAll(elem | result->count(elem) = 1)

**set->asBag : Bag(T)** The Bag that contains all the elements from set.
**post**: result->forAll(elem | set->includes(elem))
**post**: set->forAll(elem | result->count(elem) = 1)

### 12.3.3 Bag

A bag is a collection with duplicates allowed. That is, one object can be an element of a bag many times. There is no ordering defined on the elements in a bag.

Properties of Bag, where the instance of Bag is called bag.

**bag = (bag2: Bag(T)) : Boolean** True if bag and bag2 contain the same elements, the same number of times.
**post**: result = (bag->forAll(elem | bag->count(elem) =
bag2->count(elem)) and
bag2->forAll(elem | bag2->count(elem) = bag->count(elem)) )

**bag->union(bag2 : Bag(T)) : Bag(T)** The union of bag and bag2.
**post**: result->forAll( elem | result->count(elem) = bag->count(elem) + bag2->count(elem))
**post**: bag->forAll( elem | result->count(elem) = bag->count(elem) + bag2->count(elem))
**post**: bag2->forAll( elem | result->count(elem) = bag->count(elem) + bag2->count(elem))

**bag->union(set : Set(T)) : Bag(T)** The union of bag and set.
**post**: result->forAll(elem | result->count(elem) = bag->count(elem) + set->count(elem))
**post**: bag->forAll(elem | result->count(elem) = bag->count(elem) + set->count(elem))
**post**: set->forAll(elem | result->count(elem) = bag->count(elem) + set->count(elem))

**bag->intersection(bag2: Bag(T)): Bag(T)** The intersection of bag and bag2.
**post**: result->forAll(elem | result->count(elem) =
bag->count(elem).min(bag2->count(elem)) )
**post**: bag->forAll(elem | result->count(elem) =
bag->count(elem).min(bag2->count(elem)) )
**post**: bag2->forAll(elem | result->count(elem) =
bag->count(elem).min(bag2->count(elem)) )

**bag->intersection(set : Set(T)) : Set(T)** The intersection of bag and set.
**post**: result->forAll(elem | result->count(elem) =
bag->count(elem).min(set->count(elem)) )
**post**: bag->forAll(elem | result->count(elem) =
bag->count(elem).min(set->count(elem)) )
**post**: set->forAll(elem | result->count(elem) =
bag->count(elem).min(set->count(elem)) )

**bag->including(object : T) : Bag(T)** The bag containing all elements of bag plus object.
**post**: result->forAll(elem | if elem = object then result->count(elem) = bag->count(elem) + 1 else result->count(elem) = bag->count(elem) endif)
**post**: bag->forAll(elem | if elem = object then result->count(elem) = bag->count(elem) + 1 else result->count(elem) = bag->count(elem) endif)

**bag->excluding(object : T) : Bag(T)** The bag containing all elements of bag apart from all occurrences of object.
**post**: result->forAll(elem | if elem = object then result->count(elem) = 0 else result->count(elem) = bag->count(elem) endif)

**post**: bag->forAll(elem | if elem = object then result->count(elem) = 0 else result->count(elem) = bag->count(elem) endif)

**bag->select(expr : OclExpression) : Bag(T)** The sub-bag of bag for which expr is true.
**post**: result = bag->iterate(elem; acc : Bag(T) = Bag | if expr then acc->including(elem) else acc endif)

**bag->reject(expr : OclExpression) : Bag(T)** The sub-bag of bag for which expr is false.
**post**: result = bag->select(not expr)

**bag->collect(expr:OclExpression): Bag(expr.evaluationType)** The Bag of elements which results from applying expr to every member of bag.
**post**: result = bag->iterate(elem; acc : Bag(expr.evaluationType) = Bag | acc->including(expr) )

**bag->count(object : T) : Integer** The number of occurrences of object in bag.

**bag->asSequence : Sequence(T)** A Sequence that contains all the elements from bag, in undefined order.
**post**: result->forAll(elem | bag->count(elem) = result->count(elem))
**post**: bag->forAll(elem | bag->count(elem) = result->count(elem))

**bag->asSet : Set(T)** The Set containing all the elements from bag, with duplicates removed.
**post**: result->forAll(elem | bag->includes(elem) )
**post**: bag->forAll(elem | result->includes(elem))

### 12.3.4 Sequence

A sequence is a collection where the elements are ordered. An element may be part of a sequence more than once.

Properties of Sequence(T), where the instance of Sequence is called sequence.

**sequence->count(object : T) : Integer** The number of occurrences of object in sequence.

**sequence = (sequence2 : Sequence(T)) : Boolean** True if sequence contains the same elements as sequence2 in the same order.
**post**: result = Sequence1..sequence->size->forAll(index : Integer | sequence->at(index) = sequence2->at(index)) and sequence->size = sequence2->size

**sequence->union (sequence2 : Sequence(T)) : Sequence(T)**
The sequence consisting of all elements in sequence, followed by all elements in sequence2.
**post**: result->size = sequence->size + sequence2->size
**post**: Sequence1..sequence->size->forAll(index : Integer | sequence ->at(index) = result->at(index))
**post**: Sequence1..sequence2->size->forAll(index : Integer | sequence2 ->at(index) = result->at(index + sequence->size)))

**sequence->append (object:T):Sequence(T)** The sequence of elements, consisting of all elements of sequence, followed by object.
**post**: result->size = sequence->size + 1
**post**: result->at(result->size) = object
**post**: Sequence1..sequence->size->forAll(index : Integer | result->at(index) = sequence ->at(index))

**sequence->prepend(object : T) : Sequence(T)** The sequence consisting of object, followed by all elements in sequence.
**post**: result->size = sequence->size + 1
**post**: result->at(1) = object
**post**: Sequence1..sequence->size->forAll(index : Integer | sequence->at(index) = result->at(index + 1))

**sequence->subSequence(lower:Integer,upper:Integer):Sequence(T)**
The sub-sequence of sequence starting at number lower, up to and including element number upper.
**pre** : 1 <= lower pre : lower <= upper
**pre** : upper <= sequence->size
**post**: result->size = upper -lower + 1
**post**: Sequencelower..upper->forAll( index | result-> at(index - lower + 1) = sequence->at(index))

**sequence->at(i : Integer) : T** The i-th element of sequence.
**pre** : i >= 1 and i <= sequence->size sequence->first : T The first

element in sequence.
**post**: result = sequence->at(1)

**sequence->last : T** The last element in sequence.
**post**: result = sequence->at(sequence->size)

**sequence->including(object:T) : Sequence(T)** The sequence containing all elements of sequence plus object added as the last element.
**post**: result = sequence.append(object)

**sequence->excluding(object: T): Sequence(T)** The sequence containing all elements of sequence apart from all occurrences of object. The order of the remaining elements is not changed.
**post**:result->includes(object) = false
**post**: result->size = sequence->size - sequence->count(object)
**post**: result = sequence->iterate(elem; acc : Sequence(T) = Sequence| if elem = object then acc else acc->append(elem) endif )

**sequence->select(expression:OclExpression): Sequence(T)** The subsequence of sequence for which expression is true.
**post**: result = sequence->iterate(elem; acc : Sequence(T) = Sequence | if expr then acc->including(elem) else acc endif)

**sequence->reject(expression:OclExpression):Sequence(T)** The subsequence of sequence for which expression is false.
**post**: result = sequence->select(not expr)

**sequence->collect(expression : OclExpression)**
**: Sequence(expression.evaluationType)**
The Sequence of elements which results from applying expression to every member of sequence.

**sequence->iterate(expr : OclExpression) : expr.evaluationType**
Iterates over the sequence. Iteration will be done from element at position 1 up until the element at the last position following the order of the sequence.

**sequence->asBag() : Bag(T)** The Bag containing all the elements from sequence, including duplicates.
**post**: result->forAll(elem | sequence->count(elem) =

result->count(elem) )
**post**: sequence->forAll(elem | sequence->count(elem) =
result->count(elem) )

**sequence->asSet() : Set(T)** The Set containing all the elements from sequence, with duplicated removed.
**post**: result->forAll(elem | sequence->includes(elem))
**post**: sequence->forAll(elem | result->includes(elem))

## 12.4 Special Types

### 12.4.1 OclType

All types defined in a UML model, or pre-defined within OCL, have a type. This type is an instance of the OCL type called OclType. Access to this type allows the modeler limited access to the meta-level of the model. This can be useful for advanced modelers.

Properties of OclType, where the instance of OclType is called type.

**type.name : String** The name of type.

**type.attributes : Set(String)** The set of names of the attributes of type, as they are defined in the model.

**type.associationEnds : Set(String)** The set of names of the navigable associationEnds of type, as they are defined in the model.

**type.operations : Set(String)** The set of names of the operations of type, as they are defined in the model.

**type.supertypes : Set(OclType)** The set of all direct supertypes of type.
**post**: type.allSupertypes->includesAll(result)

**type.allSupertypes : Set(OclType)** The transitive closure of the set of all supertypes of type.

**type.allInstances : Set(type)** The set of all instances of type and all its subtypes in existence at the snapshot at the time that the expression is evaluated.

### 12.4.2 OclAny

Within the OCL context, the type OclAny is the supertype of all types in the model and the basic predefined OCL type. The predefined OCL Collection types are not subtypes of OclAny. Properties of OclAny are available on each object in all OCL expressions. All classes in a UML model inherit all properties defined on OclAny. To avoid name conflicts between properties in the model and the properties inherited from OclAny, all names on the properties of OclAny start with ocl. Although theoretically there may still be name conflicts, they can be avoided. One can also use the oclAsType() operation to explicitly refer to the OclAny properties.

Properties of OclAny, where the instance of OclAny is called object.

**object = (object2 : OclAny) : Boolean** True if object is the same object as object2.

**object <> (object2 : OclAny) : Boolean** True if object is a different object from object2.
**post**: result = not (object = object2)

**object.oclIsKindOf(type : OclType) : Boolean** True if type is one of the types of object, or one of the supertypes (transitive) of the types of object.

**object.oclIsTypeOf(type : OclType) : Boolean** True if type is equal to one of the types of object.

**object.oclAsType(type : OclType) : type** Results in object, but now of known type type. Results in Undefined if the actual type of object is not type or one of its subtypes.
**pre** : object.oclIsKindOf(type)
**post**: result = object post: result.oclIsKindOf(type)

**object.oclInState(state : OclState) : Boolean** Results in true if object is in the state state, otherwise results in false. The argument is a name of a state in the state machine corresponding with the class of object.

**object.oclIsNew : Boolean** Can only be used in a postcondition. Evaluates to true if the object is created during performing the operation. I.e. it didn t exist at precondition time.

### 12.4.3 OclState

The type OclState is used as a parameter for the operation oclInState. There are no properties defined on OclState. One can only specify an OclState by using the name of the state, as it appears in a statemachine. These names can be fully qualified by the nested states and statemachine that contain them.

### 12.4.4 OclExpression (Not supported in Draft Standard

Each OCL expression itself is an object in the context of OCL. The type of the expression is OclExpression. This type and its properties are used to define the semantics of properties that take an expression as one of their parameters: select, collect, forAll, etc. An OclExpression includes the optional iterator variable and type and the optional accumulator variable and type.

Properties of OclExpression, where the instance of OclExpression is called expression.

**expression.evaluationType : OclType** The type of the object that results from evaluating expression.

# Chapter 13

# Appendix: Attribute Grammar for OCL

Revised submission, Version 1.5, June 3, 2002.

## ExpressionInOclCS

The ExpressionInOcl symbol has been added to setup the initial environment of an expression.

ExpressionInOclCS ::= OclExpressionCS

**Abstract syntax mapping**
ExpressionInOclCS.ast : OclExpression

**Synthesized attributes**
ExpressionInOclCS.ast = OclExpressionCS.ast

**Inherited attributes**
The environment of the OCL expression must be defined, but what exactly needs to be in the environment depends on the context of the OCL expression. The following rule is therefore not complete. It defines the env attribute by adding the self variable to an empty environment, as well as a Namespace containing all elements visible from self. (In section 7.2 ( The ExpressionInOcl Type ) the contextualClassifier will be defined for the various places where an ocl expression may occur.) In the context of a pre- or postcondition, the result variable as well as variable definitions for any named operation para-meters can be added in a similar way.

OclExpressionCS.env = ExpressionInOclCS.contextualClassifier.namespace.getEnvironmentWit
.addElement ('self',ExpressionInOclCS.contextualClassifier,true)

## OclExpressionCS

An OclExpression has several production rules, one for each subclass of OclExpression. Note that Unspecified-ValueExp is handled explicitly in OclMessageArgCS, because that is the only place where it is allowed.

**A** OclExpressionCS :::= PropertyCallExpCS

**B** OclExpressionCS ::= VariableExpCS

**C** OclExpressionCS ::= LiteralExpCS

**D** OclExpressionCS ::= LetExpCS

**E** OclExpressionCS ::= OclMessageExpCS

**F** OclExpressionCS ::= IfExpCS

**Abstract syntax mapping**
OclExpressionCS.ast :OclExpression

**Synthesized attributes**

**A** OclExpressionCS..ast = PropertyCallExpCS.ast

**B** OclExpressionCS..ast = VariableExpCS.ast

**C** OclExpressionCS..ast = LiteralExpCS.ast

**D** OclExpressionCS..ast = LetExpCS.ast

**E** OclExpressionCS..ast = OclMessageExpCS.ast

**F** OclExpressionCS..ast = IfExpCS.ast

**Inherited attributes**

**A** PropertyCallExpCS..env = OclExpressionCS.env

**B** VariableExpCS..env = OclExpressionCS.env

**C** LiteralExpCS..env = OclExpressionCS.env

**D** LetExpCS..env = OclExpressionCS.env

**E** OclMessageExpCS..env = OclExpressionCS.env

**F** IfExpCS..env = OclExpressionCS.env

**Disambiguating rules**
The disambiguating rules are defined in the children.

## VariableExpCS

A variable expression is just a name that refers to a variable.

VariableExpCS ::= simpleNameCS

**Abstract syntax mapping**
VariableExpCS.ast : VariableExpression

**Synthesized attributes**
VariableExpCS.ast.referredVariable = env.lookup(simpleNameCS.ast).referredElement.oclAsTy

**Inherited attributes**
–none

**Disambiguating rules**

1. simpleName must be a name of a visible VariableDeclaration in the current environment.

   env.lookup (simpleNameCS.ast).referredElement.oclIsKindOf (VariableDeclaration)

## simpleNameCS

This production rule represents a single name. No special rules are applicable. The exact syntax of a String is undefined in UML 1.4, and remains undefined in OCL 2.0. The reason for this is internationalization.

simpleNameCS ::= <String>

**Abstract syntax mapping** simpleNameGr.ast : String

**Synthesized attributes** simpleNameGr.ast = <String>

**Inherited attributes** –none

**Disambiguating rules** –none

## pathNameCS

This rule represents a path name, which is held in its ast as a sequence of Strings. pathNameCS :: =simpleNameCS ( :: pathNameCS ))?

**Abstract syntax mapping** pathNameCS.ast :Sequence(String)

**Synthesized attributes**
pathNameCS.ast = SequencesimpleNameCS.ast− >union(pathNameCS.ast)

**Inherited attributes** –none

**Disambiguating rules** –none

## LiteralExpCS

This rule represents literal expressions.

**A** LiteralExpCS :::= EnumLiteralExpCS

**B** LiteralExpCS :::= CollectionLiteralExpCS

**C** LiteralExpCS :::= TupleLiteralExpCS

**D** LiteralExpCS :::= PrimitiveLiteralExpCS

**Abstract syntax mapping**
LiteralExpCS.ast :LiteralExp

**Synthesized attributes**

**A** LiteralExpCS.ast = EnumLiteralExpCS.ast

**B** LiteralExpCS.ast = CollectionLiteralExpCS.ast

**C** LiteralExpCS.ast = TupleLiteralExpCS.ast

**D** LiteralExpCS.ast = PrimitiveLiteralExpCS.ast

**Inherited attributes**

**A** EnumLiteralExpCS.env = LiteralExpCS.env

**B** CollectionLiteralExpCS.env = LiteralExpCS.env

**C** TupleLiteralExpCS.env = LiteralExpCS.env

**D** PrimitiveLiteralExpCS.env = LiteralExpCS.env

**Disambiguating rules** –none

## EnumLiteralExpCS

The rule represents Enumeration Literal expressions.

EnumLiteralExpCS ::=pathNameCS :: simpleNameCS

**Abstract syntax mapping**
EnumLiteralExpCS.ast :EnumLiteralExp

**Synthesized attributes**
EnumLiteralExpCS.ast.type = env.lookupPathName (pathNameCS.ast).referredElement.oclAsType (Classifier)

EnumLiteralExpCS.ast.referredEnumLiteral = EnumLiteralExpCS.ast.type.oclAsType (Enumeration).literal− >select(l | l.name =simpleNameCS.ast )− >any(true)

**Inherited attributes** –none

**Disambiguating rules**

1. The specified name must indeed reference an enumeration:

   not EnumLiteralExpCS.ast.type.oclIsUndefined()and EnumLiteralExpCS.ast.type.oclIsKindOf (Enumeration)

## CollectionLiteralExpCS

This rule represents a collection literal expression.

CollectionLiteralExpCS ::=CollectionTypeIdentifierCS '{' CollectionLiteralPartsCS? '}'

**Abstract syntax mapping**
CollectionLiteralExpCS.ast : CollectionLiteralExp

**Synthesized attributes**
CollectionLiteralExpCS.ast.parts = CollectionLiteralPartsCS.ast

CollectionLiteralExpCS.ast.kind = CollectionTypeIdentifierCS.ast

**Inherited attributes**
CollectionTypeIdentifierCS.env = CollectionLiteralExpCS.env

CollectionLiteralPartsCS.env = CollectionLiteralExpCS.env

**Disambiguating rules**

1. In a literal the collectiuon type may not be Collection CollectionTypeIdentifierCS.ast <> 'Collection'

## CollectionTypeIdentifierCS

This rule represent the type indentifier in a collection literal expression. The Collection type is an abstract type on M1 level, so it has no corresponding literals.

**A** CollectionTypeIdentifierCS ::= 'Set'

**B** CollectionTypeIdentifierCS ::= 'Bag'

**C** CollectionTypeIdentifierCS ::= 'Sequence'

**D** CollectionTypeIdentifierCS ::= 'Collection'

**Abstract syntax mappings**
CollectionTypeIdentifierCS.ast :CollectionKind

**Synthesized attributess**

**A** CollectionTypeIdentifierCS..ast =CollectionKind::Set

**B** CollectionTypeIdentifierCS..ast =CollectionKind::Bag

**C** CollectionTypeIdentifierCS..ast =CollectionKind::Sequence

**D** CollectionTypeIdentifierCS..ast =CollectionKind::Collection

**Inherited attributess** –none

**Disambiguating ruless** –none

## CollectionLiteralPartsCS

This production rule describes a sequence of items that are the contents of a collection literal.

CollectionLiteralPartsCS [1] == CollectionLiteralPartCS ( ',' CollectionLiteralPartsCS [2] ))?

**Abstract syntax mapping**
CollectionLiteralPartsCS [1].ast : Sequence(CollectionLiteralPart)

**Synthesized attributes**
CollectionLiteralPartsCS [1].ast = SequenceCollectionLiteralPartCS.ast− >union(CollectionLit
[2].ast)

**Inherited attributes**
CollectionLiteralPartCS.env = CollectionLiteralPartsCS [1].env

CollectionLiteralPartSCS [2].env = CollectionLiteralPartsCS [1].env

**Disambiguating rules** –none

## CollectionLiteralPartCS

**A** CollectionLiteralPartCS :::= CollectionRangeCS

**B** CollectionLiteralPartCS :::= OclExpressionCS

**Abstract syntax mapping**
CollectionLiteralPartCS.ast : CollectionLiteralPart

**Synthesized attributes**

**A** CollectionLiteralPartCS.ast = CollectionRange.ast

**B** CollectionLiteralPartCS.ast.oclIsKindOf(CollectionItem) and CollectionLiteralPartCS.ast.oclAsType(CollectionItem).OclExpression = OclExpressionCS.ast

**Inherited attributes**

**A** CollectionRangeCS..env = CollectionLiteralPartCS.env

**B** OclExpressionCS..env = CollectionLiteralPartCS.env

**Disambiguating rules** –none

## CollectionRangeCS

CollectionRangeCS ::=OclExpressionCS[1] '..' OclExpressionCS[2]

**Abstract syntax mapping**
CollectionRangeCS.ast : CollectionRange

**Synthesized attributes**
CollectionRangeCS.ast.first = OclExpressionCS[1].ast

CollectionRangeCS.ast.last =OclExpressionCS[2].ast

**Inherited attributes**
OclExpressionCS[1].env =CollectionRangeCS.env

OclExpressionCS[2].env =CollectionRangeCS.env

**Disambiguating rules** –none

## PrimitiveLiteralExpCS

This includes Real, Boolean, Integer and String literals. Exprecially String literals must take internationalisation into account and might need to remain undefined in this specification.

**A** PrimitiveLiteralExpCS :::= IntegerLiteralExpCS

**B** PrimitiveLiteralExpCS :::= RealLiteralExpCS

**C** PrimitiveLiteralExpCS :::= StringLiteralExpCS

**D** PrimitiveLiteralExpCS :::= BooleanLiteralExpCS

**Abstract syntax mapping**
PrimitiveLiteralExpCS.ast :PrimitiveLiteralExp

**Synthesized attributes**

**A** PrimitiveLiteralExpCS.ast = IntegerLiteralExpCS.ast

**B** PrimitiveLiteralExpCS.ast = RealLiteralExpCS.ast

**C** PrimitiveLiteralExpCS.ast = StringLiteralExpCS.ast

**D** PrimitiveLiteralExpCS.ast = BooleanLiteralExpCS.ast

**Inherited attributes** –none

**Disambiguating rules** –none

## TupleLiteralExpCS

This rule represents tuple literal expressions.

TupleLiteralExpCS ::= 'Tuple' '{' variableDeclarationListCS '}'

**Abstract syntax mapping**
TupleLiteralExpCS.ast :TupleLiteralExp

**Synthesized attributes**
TupleLiteralExpCS.tuplePart = variableDeclarationListCS.ast

**Inherited attributes**
variableDeclarationListCS[1].env = TupleLiteralExpCS.env

**Disambiguating rules**

1. The initExpression and type of all VariableDeclarations must exist. TupleLiteralExpCS.tuplePart− >forAll(varDecl *mid* varDecl.initExpression− >notEmpty()and not varDecl.type.oclIsUndefined())

## IntegerLiteralExpCS

This rule represents integer literal expressions.

IntegerLiteralExpCS ::= <String>

**Abstract syntax mapping**
IntegerLiteralExpCS.ast : IntegerLiteralExp

**Synthesized attributes**
IntegerLiteralExpCS.ast.integerSymbol = <String>.toInteger()

**Inherited attributes** –none

**Disambiguating rules** –none

## RealLiteralExpCS

This rule represents real literal expressions.

RealLiteralExpCS ::= <String>

**Abstract syntax mapping**
RealLiteralExpCS.ast : RealLiteralExp

**Synthesized attributes**
RealLiteralExpCS.ast.realSymbol = <String>.toReal()

**Inherited attributes** –none

**Disambiguating rules** –none

## StringLiteralExpCS

This rule represents string literal expressions.

StringLiteralExpCS ::= "'<String>"'

**Abstract syntax mappings**
StringLiteralExpCS.ast : StringLiteralExp

**Synthesized attributess**
StringLiteralExpCS.ast.symbol = <String>

**Inherited attributess** –none

**Disambiguating ruless** –none

## BooleanLiteralExpCS

This rule represents boolean literal expressions.

**A** BooleanLiteralExpCS :::= 'true'

**B** BooleanLiteralExpCS :::= 'false'

**Abstract syntax mapping**
BooleanLiteralExpCS.ast : BooleanLiteralExp

**Synthesized attributes**

**A** BooleanLiteralExpCS.ast.booleanSymbol = true

**B** BooleanLiteralExpCS.ast.booleanSymbol = false

**Inherited attributess** –none

**Disambiguating ruless** –none

## PropertyCallExpCS

This rule represents property call expressions.

**A** PropertyCallExpCS ::: =ModelPropertyCallExpCS

**B** PropertyCallExpCS :::= LoopExpCS

**Abstract syntax mapping**
PropertyCallExpCS.ast : PropertyCallExp

**Synthesized attributes**

**A** PropertyCallExpCS.ast = ModelPropertyCallCS.ast

**B** PropertyCallExpCS.ast = LoopExpCS.ast

**Inherited attributes**

**A** ModelPropertyCallCS.env = PropertyCallExpCS.env

**B** LoopExpCS.env = PropertyCallExpCS.env

**Disambiguating rules**
The disambiguating rules are defined in the children.

## LoopExpCS

This rule represents loop expressions.

**A** LoopExpCS :::=IteratorExpCS

**B** LoopExpCS :::=IterateExpCS

**Abstract syntax mapping**
LoopExpCS.ast : LoopExp

**Synthesized attributes**

**A** LoopExpCS.ast = IteratorExpCS.ast

**B** LoopExpCS.ast = IterateExpCS.ast

**Inherited attributes**

**A** IteratorExpCS.env = LoopExpCS.env

**B** IterateExpCS.env = LoopExpCS.env

**Disambiguating rules** –none

## IteratorExpCS

The first alternative is a straightforward Iterator expression, with optional iterator variable. The second and third alternatives are so-called implicit collect iterators. B is for operations and C for attributes, D for navigations and E for associationclasses.

**A** IteratorExpCS :::=OclExpressionCS [1 ] '− >' simpleNameCS '(' ((VariableDeclarationCS[1], ( ',' VariableDeclarationCS[2])? '|' ))? OclExpressionCS[2] ')'

**B** IteratorExpCS ::: =OclExpressionCS '.' simpleNameCS ( argumentsCS? )

**C** IteratorExpCS ::: =OclExpressionCS '.' simpleNameCS

**D** IteratorExpCS ::: =OclExpressionCS '.' simpleNameCS ( '[' argumentsCS ']' )?

**E** IteratorExpCS ::: =OclExpressionCS '.' simpleNameCS ( '[' argumentsCS ']' )?

**Abstract syntax mapping**
IteratorExpCS.ast : IteratorExp

**Synthesized attributes**
–the ast needs to be determined bit by bit,first the source association of IteratorExp

**A** IteratorExpCS.ast.source = OclExpressionCS [1].ast
–next the iterator association of IteratorExp

# Chapter 14

# Appendix: Zermelo-Fraenkel Axiom System

**A1 Extensionality**

$$\forall z(z \in x \leftrightarrow z \in y) \rightarrow x = y.$$

**A2 Foundation**

$$\exists y(y \in x) \rightarrow \exists y(y \in x \wedge \forall z \neg(z \in x \wedge z \in y)).$$

**A3 Subset**

$$\exists y \forall z(z \in y \leftrightarrow z \in x \wedge \phi(z)).$$

for any formula $\phi$ not containing $y$.

**A4 Empty set**

$$\exists y \forall x(x \notin y).$$

**A5 Pair set**

$$\exists y \forall x(x \in y \leftrightarrow x = z_1 \vee x = z_2).$$

**A6 Power set**

$$\exists y \forall z(z \in y \leftrightarrow \forall u(u \in z \rightarrow u \in x)).$$

**A7 Sum**

$$\exists y \forall z(z \in y \leftrightarrow \exists u(z \in u \wedge u \in x))$$

**A8 Infinity**

$$\begin{array}{ll} \exists w(\emptyset \in w \wedge & \forall x(x \in w \rightarrow \\ & \exists z(z \in w \wedge \\ & \forall u(u \in z \leftrightarrow u \in x \vee u = x)))) \end{array}$$

**A9 Replacement**

$$\begin{array}{l} \forall x, y, z(\psi(x, y) \wedge \psi(x, z) \rightarrow y = z) \rightarrow \\ \exists u \forall w_1(w_1 \in u \leftrightarrow \exists w_2(w_2 \in a \wedge \psi(w_2, w_1))) \end{array}$$

**A10 Axiom of Choice**

$$\begin{array}{l} \forall x(x \in z \rightarrow x \neq \emptyset \wedge \\ \forall y(y \in z \rightarrow x \cap y = \emptyset \vee x = y)) \\ \rightarrow \\ \exists u \forall x \exists v(x \in z \rightarrow u \cap x = \{v\}) \end{array}$$

# Chapter 15

# Appendix: Axiom Systems for Sequent Calculi

## 15.1 The Axiom System $S_0$

**axiom**

$$\frac{}{\Gamma, F \rightarrow F, \Delta}$$

**not-left**

$$\frac{\Gamma, \rightarrow F, \Delta}{\Gamma, \neg F \rightarrow \Delta}$$

**not-right**

$$\frac{\Gamma, F \rightarrow \Delta}{\Gamma \rightarrow \neg F, \Delta}$$

**impl-left**

$$\frac{\Gamma \rightarrow F, \Delta \quad \Gamma, G \rightarrow \Delta}{\Gamma, F \rightarrow G \rightarrow \Delta}$$

**impl-right**

$$\frac{\Gamma, F \rightarrow G, \Delta}{\Gamma \rightarrow F \rightarrow G, \Delta}$$

**and-left**

$$\frac{\Gamma, F, G \rightarrow \Delta}{\Gamma, F \wedge G \rightarrow \Delta}$$

**and-right**

$$\frac{\Gamma \rightarrow F, \Delta \quad \Gamma \rightarrow G, \Delta}{\Gamma \rightarrow F \wedge G, \Delta}$$

**or-left**

$$\frac{\Gamma, F \rightarrow \Delta \quad \Gamma, G \rightarrow \Delta}{\Gamma, F \vee G \rightarrow \Delta}$$

**or-right**

$$\frac{\Gamma \rightarrow F, G, \Delta}{\Gamma \rightarrow F \vee G, \Delta}$$

**all-left**

$$\frac{\Gamma, \forall x F, F(t/x) \rightarrow \Delta}{\Gamma, \forall x F \rightarrow \Delta}$$

where $t$ is a ground term.

**all-right**

$$\frac{\Gamma \rightarrow F(c/x), \Delta}{\Gamma \rightarrow \forall x F, \Delta}$$

where $c$ is a new constant symbol.

**ex-right**

$$\frac{\Gamma \rightarrow \exists x F, F(t/x), \Delta}{\Gamma, \rightarrow \exists x F, \Delta}$$

where $t$ is ground term.

**ex-left**

$$\frac{\Gamma \rightarrow F(c/x), \Delta}{\Gamma, \exists x F \rightarrow \Delta}$$

where $c$ is a new constant symbol.

## 15.2 The Axiom System $S_0^{fv}$

**axiom**

$$\frac{}{\Gamma, F \rightarrow F, \Delta}$$

**not-left**

$$\frac{\Gamma, \rightarrow F, \Delta}{\Gamma, \neg F \rightarrow \Delta}$$

**not-right**

$$\frac{\Gamma, F \rightarrow \Delta}{\Gamma \rightarrow \neg F, \Delta}$$

**impl-left**

$$\frac{\Gamma \rightarrow F, \Delta \quad \Gamma, G \rightarrow \Delta}{\Gamma, F \rightarrow G \rightarrow \Delta}$$

**impl-right**

$$\frac{\Gamma, F \rightarrow G, \Delta}{\Gamma \rightarrow F \rightarrow G, \Delta}$$

**and-left**

$$\frac{\Gamma, F, G \rightarrow \Delta}{\Gamma, F \wedge G \rightarrow \Delta}$$

**and-right**

$$\frac{\Gamma \rightarrow F, \Delta \quad \Gamma \rightarrow G, \Delta}{\Gamma \rightarrow F \wedge G, \Delta}$$

**or-left**

$$\frac{\Gamma, F \rightarrow \Delta \quad \Gamma, G \rightarrow \Delta}{\Gamma, F \vee G \rightarrow \Delta}$$

**or-right**

$$\frac{\Gamma \rightarrow F, G, \Delta}{\Gamma \rightarrow F \vee G, \Delta}$$

**all-left**

$$\frac{\Gamma, \forall x F, F(X/x) \rightarrow \Delta}{\Gamma, \forall x F \rightarrow \Delta}$$

where $X$ is a new variable.

**all-right**

$$\frac{\Gamma \rightarrow F(f(x_1, \ldots, x_n)/x), \Delta}{\Gamma \rightarrow \forall x F, \Delta}$$

where $f$ is a new functions symbol and $x_1, \ldots, x_n$ are all free variables in $\forall x F$.

**ex-right**

$$\frac{\Gamma \rightarrow \exists x F, F(X/x), \Delta}{\Gamma, \rightarrow \exists x F, \Delta}$$

where $X$ is a new variable.

**ex-left**

$$\frac{\Gamma \rightarrow F(f(x_1, \ldots, x_n)/x), \Delta}{\Gamma, \exists x F \rightarrow \Delta}$$

where $f$ is a new functions symbol and $x_1, \ldots, x_n$ are all free variables in $\forall x F$.

## 15.3 Rules for Equality

**identity-right**

$$\frac{}{\Gamma, \rightarrow s = s, \Delta}$$

**identity-left**

$$\frac{}{\Gamma, s = s \rightarrow \Delta}$$

**symmetry-right**

$$\frac{\Gamma \rightarrow s = t, \Delta}{\Gamma \rightarrow t = s, \Delta}$$

**symmetry-left**

$$\frac{\Gamma, s = t \rightarrow \Delta}{\Gamma, t = s \rightarrow \Delta}$$

**eq-subst-right**

$$\frac{\Gamma, s = t \rightarrow F(t), \Delta}{\Gamma, s = t \rightarrow F(s), \Delta}$$

**eq-subst-left**

$$\frac{\Gamma, F(t), s = t \rightarrow \Delta}{\Gamma, F(s), s = t \rightarrow \Delta}$$

# Chapter 16

# Appendix: Source Code

## 16.1 Java Programs

```
import java.io.*;

public class RussM{

static int a;
static int b;
static int z = 0;

public static void main(String[] args){

   System.out.println("Erste Zahl?"); a = LiesInt();
   System.out.println("Zweite Zahl?"); b = LiesInt();

   while (b!=0){
     if  ((b / 2) * 2 == b)
       {a = 2*a;
        b = b / 2;}
      else
        {z = z + a;
         a = 2*a;
          b = b / 2;
   }}
    System.out.print("Ergebnis: ");
   System.out.println(z);
       }

 static int LiesInt() {

  DataInput StdEingabe = new DataInputStream(System.in);
  int ergebnis = 0;
    try{ ergebnis = Integer.parseInt(StdEingabe.readLine()); }
    catch (IOException io) {}
    return ergebnis;
  }}
```

```
import java.io.*;

public class Text { public int len;
                    public int [] seq;

public static void main(String[] args){ int ll; int pos;

        System.out.print("sequence lenght? "); ll = LiesInt();
        Text x = new Text();
        x.len = ll;
        x.seq = new int[ll];
        readAll(ll,x); PrintAll(x);
        System.out.print("delete position? "); pos = LiesInt();
        x.delete(pos,ll);
        PrintAll(x);}

public void delete(int p, int l){
        for (int a = p; a < l-1 ; a = a +1)
        {this.seq[a] = this.seq[a+1]; }
        this.len = this.len - 1; }

static void readAll(int yy, Text obj) { int in;
  for (int a = 0; a < yy ; a = a +1)
  {System.out.print((a+1)+"-th entry? "); in = LiesInt();
  obj.seq[a] = in; }}

static void PrintAll(Text obj) { int y; y = obj.len;
        System.out.println();
        for (int a = 0; a < y ; a = a +1){
             System.out.print(obj.seq[a]+" ");}
        System.out.println();}

 static int LiesInt() {
  DataInput StdEingabe = new DataInputStream(System.in);
  int ergebnis = 0;
   try{ ergebnis = Integer.parseInt(StdEingabe.readLine()); }
   catch (IOException io) {}
  return ergebnis;}}
```

## 16.2 KeYProver Input

### 16.2.1 Induction Proof Task

```
sorts { int;}

schema variables {int variables x,y,z;}

functions {int 0;
           int a(int,int);
           int f(int);}
problem{
   all y:int.(a(0,y)=y) &
   all x:int. all y:int. (a(f(x),y) = f(a(x,y)))  &
   all x:int. all y:int.(a(succ(x),y) = f(a(x,y)))
-> all x:int.( geq(x,0) -> a(x,a(x,x)) = a(a(x,x),x))
     }
```

# Bibliography

[ACM, 1999] ACM, editor. *Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '99*, volume 34 (10) of *ACM SIGPLAN notices*. ACM, ACM Press, 1999.

[Álvarez & Alemán, 2000] Ambrosio Toval Álvarez & José Luis Fernández Alemán. Formally modeling UML and its evolution: A holistic approach. In Smith & Talcott [Smith & Talcott, 2000], pages 183–206.

[Apt & Olderog, 1991] Krzysztof R. Apt & Ernst-Rüdiger Olderog. *Verification of sequential and concurrent programs*. Texts and Monographs in Computer Science. Springer, 1991.

[Apt, 1981] K. R. Apt. Ten years of Hoare logic: A survey - part I. *ACM Trans. on Prog. Languages and Systems*, 1981.

[Beckert *et al.*, 2004] Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Andreas Roth, Philipp Rümmer, & Steffen Schlager. Taclets: A new paradigm for constructing interactive theorem provers. *RACSAM, Rev. R. Acad. Cien. Serie A. Mat.*, page to appear, 2004.

[Bittner & Koch, 2000] Margot Bittner & Wilfried Koch. Objektorientierte analyse und design. die fusion-methode unter verwendung von UML. Technischer bericht, Institut f"ur Kommunikation - und Softwaretechnik, TU Berlin, Franklinstr. 28/29, D-10587 Berlin, 2000.

[Boldsoft *et al.*, 2002] Boldsoft, Rational Software Co, & IONA. Response to the uml 2.0 ocl rfp. revised submission. (omg document ad/2002-05-09)., June 2002.

[Bosch & Mitchell, 1998] Jan Bosch & Stuart Mitchell, editors. *Object-oriented technology : ECOOP '97 workshop reader*, volume 1357 of *LNCS*, Jyväskylä, Finalnd, 1998. Springer.

[Breu *et al.*, 1998] Ruth Breu, Radu Gosu, Franz Huber, Bernhard Rumpe, & Wolfgang Schwerin. Towards a precise semantics for object-oriented modeling techniques. In Bosch & Mitchell [Bosch & Mitchell, 1998], pages 205–210.

[Cantor, 1895] Georg Cantor. Beiträge zur Begründung der transfiniten Mengenlehre, I. *Math. Annals*, 46:481–512, 1895. A translation into English by P. E. B Jourdain has been published by Dover Publications Inc in 1915.

[Clark & Warmer, 2001] Tony Clark & Jos Warmer, editors. *Advances in Object Modelling with the OCL*, volume 2263 of *LNCS*. Springer Verlag, 2001.

[Clark & Warmer, 2002] Tony Clark & Jos Warmer, editors. *Object Modeling with the OCL. The Rationale behind the Object Constraint Language*, volume 2263 of *LNCS*. Springer, 2002.

[Clark *et al.*, 2000] Tony Clark, Andy Evans, & Stuart Kent. A feasibility study in rearchitecting UML as a family of languages using a precise OO meta-modeling approach. Technical report, pUML group and IBM, 2000.

[Cohen, 1998] B. Cohen. Set theory as a semantic framework for object-oriented modeling. In Bosch & Mitchell [Bosch & Mitchell, 1998], pages 161–168.

[Distefano *et al.*, 2000] Dino Distefano, Joost-Pieter Katoen, & Arend Rensink. Towards model checking OCL. In *Proceedings, ECOOP Workshop on Defining a Precise Semantics for UML*, 2000.

[Evans *et al.*, 1999] Andy Evans, Robert France, & Emanuel Grant. Towards formal reasoning with uml models. In *Proceedings of the OOPSLA'99 Workshop on Behavioral Semantics*, 1999.

[Ferraiolo *et al.*, 2000] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, R. Richard Kuh, & Ramaswamy Chandramouli. A proposed standard for role-based-access control. NIST, December 2000.

[Fitting & Mendelsohn, 1999] Melvin Fitting & Richard L. Mendelsohn. *First-Order Modal Logic*, volume 277 of *Synthese Library*. Kluwer Academic Publishers, 1999.

[Fowler & Scott, 1997] Martin Fowler & Kendall Scott. *UML Destilled. Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.

[France & Rumpe, 1999] Robert France & Bernhard Rumpe, editors. *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*. Springer, 1999.

[France, 1999] Robert France. A problem-oriented analysis of basic uml static modeling concepts. In ACM [ACM, 1999].

[Gallier, 1986] Jean H. Gallier. *Logic for Computer Science: Foundations of Automated Theorem Proving*. Harper and Row, New York, 1986.

[Gamma *et al.*, 1995] Erich Gamma, Richard Helm, Ralph Johnson, & John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading/MA, 1995.

[Gogolla & Richters, 1998] Martin Gogolla & Mark Richters. On constraints and queries in UML. In Schader & Korthaus [Schader & Korthaus, 1998], pages 109–121.

[Gogolla & Richters, 2002] Martin Gogolla & Mark Richters. Expressing UML Class Diagrams Properties with OCL. In Clark & Warmer [Clark & Warmer, 2002], pages 86–115.

[Gurevich, 1984] Yuri Gurevich. 'reconsidering turing's thesis: Toward more realistic semantics of programs. Technical Report CRL-TR-36-84, University of Michigan, Computing Research Lab, 1984.

[Gurevich, 2000] Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.

[Halmos, 1974] Paul Richard Halmos. *Naive Set Theory*. Springer-Verlag Telos, 1974.

[Halmos, 1994] Paul Richard Halmos. *Naive Mengenlehre*. Vandenhoeck & Ruprecht, 1994.

[Hamie *et al.*, 1998] Ali Hamie, John Howse, & Stuart Kent. Interpreting the object constraint language. In *Proceedings of Asia Pacific Conference in Software Engineering*. IEEE Press, July 1998.

[Hamie, 1998] Ali Hamie. A formal semantics for checking and analysing UML models. In Luis Andrade, Ana Moreira, Akash Deshpande, & Stuart Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998.

[Harel *et al.*, 2000] David Harel, Dexter Kozen, & Jerzy Tiuryn. *Dynamic Logic*. The MIT Press, 2000.

[Harel, 1984] David Harel. Dynamic logic. In *Handbook of Philosophical Logic*, volume II Extensions of Classical Logic, pages 497 – 604. D.Reidel, 1984.

[Hennicker *et al.*, 2001] Rolf Hennicker, Heinrich Hussmann, & Michel Bidoit. On the precise meaning of OCL constraints. In Clark & Warmer [Clark & Warmer, 2001].

[Kim & Carrington, 1999] Soon-Kyeong Kim & David Carrington. Formalizing the UML class diagram using Object-Z. In France & Rumpe [France & Rumpe, 1999], pages 83–98.

[Kozen & Tiuryn, 1990] Dexter Kozen & Jerzy Tiuryn. Logics of programs. In van Leeuwen [van Leeuwen, 1990], chapter 14, pages 791–840.

[Krishnan, 2000] P. Krishnan. Consistency Checks for UML. In *Proceedings of the Asia Pacific Software Engineering Conference (APSEC 2000)*, pages 162–169, December 2000.

[Menzel & Schmitt, 2001] Wolfram Menzel & Peter H. Schmitt. Formale systeme. Vorlesungsskript (in deutsch, 2001.

[OMG, 1999a] OMG. Object constraint language specification, version 1.3. chapter 7 in [OMG, 1999b]. OMG Document ad970808, September 1999.

[OMG, 1999b] OMG. OMG unified modeling language spezification, version 1.3. OMG Document, June 1999.

[OMG, 2000a] OMG. Object constraint language specification, version 1.3. chapter 7 in [OMG, 2000b]. OMG Document, March 2000.

[OMG, 2000b] OMG. OMG unified modeling language spezification, version 1.3. OMG Document, March 2000.

[OMG, 2001] OMG. OMG unified modeling language spezification, version 1.4 draft. OMG Document, February 2001.

[Reggio *et al.*, 2000] G. Reggio, M. Cerioli, & E. Astesiano. An algebraic semantics of UML supporting its multiview approach. In *Proc. AMiLP 2000*. University of Twente, 2000.

[Reisig, 2001] Wolfgang Reisig. Abstract state maschines. Lecture Notes (Slides), 2001.

[Rubin, 1967] Jean E. Rubin. *Set Theory for the Mathematician*. Holden-Day, 1967.

[Rumbaugh *et al.*, 1998] James Rumbaugh, Ivar Jacobson, & Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[Rumbaugh *et al.*, 1999a] James Rumbaugh, Ivar Jacobson, & Grady Booch. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[Rumbaugh *et al.*, 1999b] James Rumbaugh, Ivar Jacobson, & Grady Booch. *The Unified Software Development Process*. Addison-Wesley, 1999.

[Russo, 2001] Majorie Russo. *Java Semantics*. PhD thesis, INRIA, Unit de Sophia-Antipolis, 2001.

[Schader & Korthaus, 1998] Martin Schader & Axel Korthaus, editors. *The Unified Modeling Language: technical aspects and applications*. Physica-Verlag, 1998.

[Shilov & Yi, 2001] N. V. Shilov & K. Yi. How to find a coin: propositional program logics made easy. *Bulletin of the EATCS*, 75:127–151, October 2001.

[Smith & Talcott, 2000] Scott F. Smith & Carolyn L. Talcott, editors. *Formal Methods for open pbject-based distributed systems IV*, Standford, California, USA, September 2000. IFIP TC6/WG6.1, Kluwer Academic Publishers.

[Sperschneider & Antoniou, 1991] Volker Sperschneider & G. Antoniou. *Logic, a foundation for computer science*. Addison-Wesley, 1991.

[Takeuti & Zaring, 1971] Gaisi Takeuti & Wilson M. Zaring. *Introduction to Axiomatic Set Theory*. Graduate Texts in Mathematics. Springer, 1971.

[van Leeuwen, 1990] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume B : Formal Models and Semantics. Elsevier, Amsterdam, 1990.

[Warmer & Kleppe, 1999] Jos Warmer & Anneke Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.

# Index