A Formal Object-Oriented Specification Language without inheritance

Ondřej Ryšavý, Faculty of Information Technology, VUT Brno, Czech Republic

Abstract: *Formal specification language and deduction system for object-oriented models is proposed in this article. It extends ς-calculus introduced by Martín Abadi and Luca Cardelli with propositional connectives, and quantified terms. The first-order type system of ς-calculus serves as typing theory of suggested calculus. Deduction system consists of reduction rules of ς-terms, equality theory of ς-calculus and adapted rules of Gentzen logic system.*

1 Introduction

During the last decade the object-oriented principles and concepts has taken major place among techniques using for development of software systems. Much effort has been invested in systematizing and evolving of object-oriented techniques for every step of development process. Analytic tools, specification languages and programming languages that use object-oriented principles are currently in use. They advantages, although they can be achieved by traditional techniques as well, make object-oriented approach uniquely successful. Most of the benefits of the object-oriented approach come directly from its modularity. Each of the objects which make up a system can be understood and reasoned about in isolation. Understanding the system is then reduced to understanding the interactions between its objects. Object orientation also encourages software reuse since smaller components are more likely to be reused in a system than larger ones. It is also possible to make modification or extending of systems easier. Often simple modifications will only affect one object and they can be made to that object in isolation.

Object orientation can be used in the formal specification of software as a means of handling the complexity of large-scale systems. The structuring and reusability provided by adopting an object-oriented approach improves the clarity and also can aids during verification and refinement. Properties of objects can be verified localy and then they can be used to prove global property of the whole system. The specification can be refined represent the exact structure, in terms of attributes and operations, of each object in the implementation.

Object oriented concepts have been incorporated in several specification languages including SDL, Estelle, VDM, CSP, LOTOS and Z. Approach used in this article tries to define the simple calculus with strong expressiveness for development of object oriented formal specifications that will achieve discussed benefits of object orientation. The article is divided to several chapters that address features and benefits of object orientation, description of ς-calculus, first-order type system for ς-calculus, proposed specification language, and deduction system of specification language.

2 Object-Oriented Features

This section discusses object-oriented features, their benefits and introduces notations and abbreviations for later chapters. The exclusive position takes issue of typing for objects. The one of the most discussed attribute of object-oriented technology is reusability. The *reusability* assumes that system can often be built out of existing objects. To reuse objects there has to be mechanism to customize them. Often new objects are *derived* from old ones by taking their attributes, adding new ones and/or replacing attributes. Reuse of existing methods is called *inheritance*, replacement of methods is called *overriding*. When a new method overrides an old one it has to conform with some flexibility to the interface of old one. In

particular, the new method may be more specific than old one and may use only attributes that are available only in the derived object.
Another feature of object-orientation approach is *information hiding*. By encapsulating object's attributes or properties, an object restricts access to its information. Together with good design of object interactions the objects could be as independent of each other as the domain they represent permits. The error-less of such objects is easier proven then large complex system with many interactions between blocks. This leads to increase *reliability* of systems built from well-proven objects. In some cases is better to reuse object by mechanism of composition. *Composition* denotes the use of one object inside the other object. This object contains the whole already defined object. The effect is that complexity of the containing object is reduced.
In the second part of this section notations of object types and classes are introduced. We have not mentioned class so far, but the most of programming languages are class-based and the classes are used abundantly in design as well. A class is intended to describe structure of all objects generated from it. Class consists of attributes divided into fields and methods. Each class has a set of objects that are called instances. These objects share methods with class, but contains fields those values are independent. To access attributes inside the method of object the operator *self* is used. It refers to object, which the method is invoked for. The all instances of class have the same type. The type of object o that is an instance of a class c is denoted as *o:InstanceTypeOf(c)* rather then denote it as a name of class itself to avoid confusion between types and classes. Inheritance described above takes place in class model as a subclassing mechanism. A subclass describes the structure of a set of objects. It is done incrementally by description of extensions and changes to superclass. There are differences between fields and methods when subclassing. Fields from superclass are replicated in subclass implicitly, and new ones may be added. Methods may be replicated from superclass in subclass, new ones may be added, or explicitly overridden. Overridden methods have to preserve name and type. Subclass declaration induces partial order denoted as *subclass relation*. Note that self in a method of superclass refers to subclassing object not to instance of a class that defines the method. When method is overridden a special operator often denoting as *super* is needed to access methods of superclasses. Nevertheless this may fail when *multiple inheritance* is allowed. To enable sense use of inheritance a techniques that allows use of subclassing objects in the place where their superclass is expected. This is often called subtype polymorphism that is expressed by the rule:

(1) If c' is a sublass of c, and o':InstanceTypeOf(c'), then o':InstanceTypeOf(c)

An intuitively defined subtyping relation satisfies these two properties:

(2) If a : A and A <: B, then a : B

(3) InstanceTypeOf(c') <: InstanceTypeOf(c) iff c' is a sublass of c

Property (2) is called *subsumption* and it is a characteristic property of subtype relations. By subsumption an object a of type A can be viewed as an object of supertype B, i.e object a is subsumed from type A to type B. Property (3) assigns subtyping and subclassing. Since inheritance is connected with subclassing this property identifies it also with subtyping. This correlation between inheritance, subclassing and subtyping is typical for classical class-based approach.
The rest of the section deals with some subtyping techniques that require review of the basic subtyping properties of product and function types. The product type A×B is the type of pairs with left component of type A and right component of type B. We say that × is *covariant* operator, because A×B varies in the same sense as A or B:

$$A\times B <: A'\times B' \quad \text{provided that } A<:A' \text{ and } B<:B'$$

The type A→B is the type of function with argument type A and result type B. We say → is a *contravariant* operator in its left argument because A→B varies in the opposite sense as A.

A→B <: A'→B' provided that A'<:A and B<:B'

Finally, denote A#B as a pair of component that may be updated, i.e. one or both value of pair can be replaced. The operator # is called *invariant*, because does not enjoy any covariance or contravariance properties.

A#B <: A'#B' provided that A = A' and B = B'

All operators that have been just defined take place when we consider possible types of methods and their arguments in subclassing. So far when method has been overridden it had exactly same type as the original method. When *method specialization* is allowed for overriding, methods may get arguments and return values of adapted types according to subclass. Types of arguments vary contravariantly on subclass relation while types of results vary covariantly. Fields cannot be specialized and their types remain unchanged (invariant). These conditions assure that instance of subclass is usable in place of general one. Another form of specialization happens implicitly by inheritance. The occurrence of self within the methods of class c has type InstanceTypeOf(c). In inherited class c' is type implicitly specialized to InstanceTypeOf(c').

A classical class-based concept defines the strict correlation between inheritance, subclassing, and subtyping. In some situations the separation of types and classes is suitable. One of the most significant cases is code reuse by parameterization. Another requirement is to divide implementation from specification and allow development within large teams of programmer. This brings conflict with previously defined concept of classes. The new approach separates object specification and implementation by introducing types for objects independently of specifying classes. Object types defined by *object protocol* consist of attribute list with assigned types while classes introduce whole implementation of object. When class c has type C we use notation o: ObjectTypeOf(c) for the object o of type C, or we may write directly o: C. Different classes c and c' may produce the same object type C. Object having type C are required only to satisfy a certain protocol. Since subtype relation was based on subclass relation its definition has to be defined independently. Only structural subtyping is considered although subtyping based on types name is also possible. For two object types O and O' assume that:

O' <: O if O' has the same components as O and possible more

As a consequence of the new definition of subtyping we have:

(4) If c' is a subclass of c, then ObjectTypeOf(c') <: ObjectTypeOf(c).

This represents subclassing-is-subtyping property but in spite of property (3) this does not hold in opposite direction. There may be unrelated c and c' such that ObjectTypeOf(c) = O and ObjectTypeOf(c')=O, with O' <: O.

The use of subsumption on subclasses is still valid because subclassing implies subtyping, and, since subsumption is based only on subtyping much more freedom is given to subsumption. Further step in considering about relation between subsuming and subtyping is to make them quite independent and avoid any assumption between them. The price paid for this added flexibility in inheritance is decreased flexibility in subsumption.

In this section some general concepts was briefly discussed. Detailed principles of object-oriented design can be found for instance in []. The main part of chapter belonged to typing and its relation to notion of class. At first types and classes were strictly assigned ones to others. This approach is simple but in some cases does not allow use techniques as parametric or pure interface definitions. Relaxing correlation between types and classes brings possibility using of advantage techniques, but it restricts ability of subsumption.

3 Object calculi

In this section object calculi developed by Matín Abadi and Luca Cardelli in [] is described. An approach used to create object calculi defines simple core flexible enough to represent complex notations, but not directly contain them. This seems better than construct complex object model that express directly all the possible variations of object classification and inheritance. The object calculus represents same level abstraction as λ-calculi that became formal base of procedural languages. Several object calculi was introduced in [1] therefore all are built from the same small core. Objects are considered to be collections of methods. Fields are special type of methods that do not use self parameters. The unification of fields with methods has the advantage of simplicity in assuming of uniform object structure. The tree natural operations of object calculi are method invocation, field selection and field update. The unification of methods and fields concludes in method update. It is not usual aspect of common object-oriented languages, but it can be viewed as a form of dynamic inheritance known from object-based languages that have not been discussed here.

The untyped calculus of objects are introduced first. Object primitives and their semantic will be described. Although object calculus does not include functions, it will be showed how to write functions and fixpoint operators in term of objects.

Objects are only computational structures in object calculi. An object is a collection of named attributes. All attributes are methods. Fields are methods that do not use self in their body. Each method has a bound variable that represents self and a body that produces a result. The only operations on objects are method invocation and method update. Except for variables, calculus does not contain any other structures.

Definition 1: Syntax of ς-calculus

a, b :: =	terms
x	variable
$[l_i=\varsigma(x_i)b_i^{\,i=(1..n)}]$	object with n methods labeled $l_1,...,l_n$, labels are distinct
$a.l$	invocation of method l of object o
$a.l \leftarrow\!\!\shortmid \varsigma(x)b$	update of method of object o with method $\varsigma(x)b$

As has been introduced above fields are represented by methods that do not use variable self in their bodies. Nevertheless some abbreviations for *field* and *field update* may be used to make expressions clearly.

$[..., l=b, ...]$ stands for $[..., l =\varsigma(y)b, ...]$, where y is not used inside b. (field)

$o.l := b$ stands for $o.l \leftarrow\!\!\shortmid \varsigma(y)b$, where y is not used inside b. (field update)

Before semantic of ς-calculus can be introduced the definition of free variable and substitution has to be provided. Set of *free variables* (FV) for ς-terms is constructed according to following definition:

$FV(\varsigma(y)b)$	$\triangleq$	$FV(b) - \{y\}$
$FV(x)$	$\triangleq$	$\{x\}$
$FV([l_i= \varsigma(x_i)b_i^{\,i\in(1..n)}])$	$\triangleq$	$\cup^{\,i\in(1..n)} FV(\varsigma(x_i)b_i)$
$FV(a.l)$	$\triangleq$	$FV(a)$
$FV(a.l \leftarrow\!\!\shortmid \varsigma(x)b)$	$\triangleq$	$FV(a) \cup FV(\varsigma(y)b)$

Substitution for ς-terms is given by following definition:

$(\varsigma(y)b)\{x\leftarrow c\}$	$\triangleq$	$\varsigma(y')(b\{y\leftarrow y'\}\{x\leftarrow c\})$ for $y' \notin FV(\varsigma(y)b) \cup FV(c) \cup \{x\}$
$x\{x\leftarrow c\}$	$\triangleq$	c

$$
\begin{array}{lcll}
y\{x\leftarrow c\} & \triangleq & y & \text{for } x\neq c \\
[l_i = \varsigma(x_i)b_i{}^{i\in(1..n)}]\{x\leftarrow c\} & \triangleq & [l_i = (\varsigma(x_i)b_i)\{x\leftarrow c\}\ {}^{i\in(1..n)}] & \\
(a.l)\{x\leftarrow c\} & \triangleq & (a\{x\leftarrow c\}).l & \\
(a.l\leftarrow\!\!\shortmid\varsigma(y)b)\{x\leftarrow c\} & \triangleq & (a\{x\leftarrow c\}).l\leftarrow\!\!\shortmid((\varsigma(y)b)\{x\leftarrow c\}) &
\end{array}
$$

To make further dealing with ς-calculus easier some notations are defined. A *closed term* is a term without free variables. We write $b\{x\}$ to highlight that x may occur free in b. We identify $\varsigma(x)b$ with $\varsigma(y)(b\{x\leftarrow y\})$, for any y not occurring free in b. We identify any two objects that differ only in the order of their components.

The *primitive semantic* of calculus is defined by reduction steps. An execution of a ς-calculus term is expressed as a sequence of reduction steps. The notation $b \rightarrowtail c$ means that b is reduced to c in one step. The substitution of term c for the free occurrence of x in b is written $b\{[x\leftarrow c\}$. The primitive semantic is given as follows.

$$
\begin{array}{llll}
\text{Let } o \equiv [l_i = \varsigma(x_i)b_i{}^{i\in(1..n)}] & & & \text{object} \\
o.l_j & \rightarrowtail \ b_j[x \leftarrow o] & (j\in 1..n) & \text{invocation reduction} \\
o.l_j \leftarrow\!\!\shortmid \varsigma(y)b & \rightarrowtail \ [l_j = \varsigma(y)b,\ l_i = \varsigma(x_i)b_i{}^{i\in(1..n)-\{j\}}] & (j\in 1..n) & \text{update reduction}
\end{array}
$$

A method invocation reduces to the result of the substitution of the host object for the self parameter in the body of method. A method update reduces to a copy of host object with replaced updated method with updating one.

Similar to λ-calculus the reduction relations can be defined. It states formal what was hold for primitive semantic. The term context is introduced in the same manner as in λ-calculus.

Definition 2: Reduction relations

(1) We write $a \rightarrowtail b$ if for some $o \equiv [l_i = \varsigma(x_i)b_i\{x_i\}^{i\in(1..n)}]$ and $j\in 1..n$, either:
 $a \equiv o.l_i$ and $b \equiv b_i\{x_i \leftarrow o\}$, or
 $a \equiv o.l_j \leftarrow\!\!\shortmid \varsigma(x)c$ and $b \equiv [l_j=\varsigma(x)c,\ l_i= \varsigma(x_i)b_i\{x_i\}^{i\in(1..n)-\{j\}}]$.

(2) A context C[-] is a term with single hole, and C[d] represents the result of filling the hole with the term d (possibly capturing free variables of d).
 We write $a \rightarrow b$ if $a \equiv C[a']$, $b\equiv C[b']$, and $a'\rightarrowtail b'$, where C[-] is any context.

(3) We write $\twoheadrightarrow$ for reflexive and transitive closure of $\rightarrow$.

Church-Rosser theorem describes confluence easier property, that if an expression may be evaluated in two different ways, both will lead to the same result.

Theorem 3: Church-Rosser

If $a \twoheadrightarrow b$ and $a \twoheadrightarrow c$, then there exists d such that $b\twoheadrightarrow d$ and $c \twoheadrightarrow d$.

From untyped reduction rules an untyped equation theory is derived. Purpose of equation theory is to capture notion of equality for ς-terms. It says that two object that behave in the same way are equal under this assumptions. Rules of equation theory are given as a set of premises and a conclusion divided by horizontal line. Each judgment has form $\vdash\mathfrak{I}$.

Definition 4: Rules of Equational theory

(Eq Symm)

$$\frac{b \leftrightarrow a}{a \leftrightarrow b}$$

(Eq Trans)

$$\frac{a \leftrightarrow b \quad b \leftrightarrow c}{a \leftrightarrow c}$$

(Eq x)

$$\frac{}{x \leftrightarrow x}$$

(Eq Object)

$$\frac{a \leftrightarrow a' \quad b \leftrightarrow b'}{a.l \leftarrow \varsigma(x)b \leftrightarrow a'.l \leftarrow \varsigma(x)b'}$$

(Eval Select) (where a≡$[l_i=\varsigma(x_i)b_i\{x_i\}^{i\in 1..n}]$)

$$\frac{j \in 1..n}{a.l_j \leftrightarrow b_j\{a\}}$$

(Eval Update) (where a≡$[l_i=\varsigma(x_i)b_i\{x_i\}^{i\in 1..n}]$)

$$\frac{j \in 1..n}{a.l_j \leftarrow \varsigma(x)b_j \leftrightarrow [l_j = \varsigma(x)b_j, l_i = \varsigma(x_i)b_i^{i\in(1..n)-\{j\}}]}$$

A derivation is a tree of judgments. Each new node is obtained by application of rule to leaves of tree. A valid judgment is one that can be obtained as the root of derivation tree.
The last semantic of ς-calculus introduced here is *operational semantic*. Neither reduction relation nor equation theory do not consider evaluation order of ς-terms. The operational semantic defines a reduction system for closed ς-terms. Main idea is that when given an object $[l_i=\varsigma(x_i)b_i\{x_i\}^{i\in 1..n}]$ we defer reducing the body b_i until l_i is invoked. The purpose of reduction system is to reduce every closed expression to a *result.* A result is itself an expression of the form $[l_i=\varsigma(x_i)b_i\{x_i\}^{i\in 1..n}]$. Weak reduction relation is denoted $\rightsquigarrow$ and has to satisfy rules given by the definition 5.

Definition 5: Rules of Operational semantics

(Red Object) (where v ≡$[l_i=\varsigma(x_i)b_i^{i\in 1..n}]$)

$$\frac{}{v \to v}$$

(Red Select) (v' ≡ $[l_i=\varsigma(x_i)b_i\{x_i\}^{i\in 1..n}]$)

$$\frac{a \to v' \quad b_j\{v'\} \to v \quad j \in 1..n}{a.l_j \to v}$$

(Red Update)

$$\frac{a \to [l_i = \varsigma(x_i)b_i^{i\in 1..n}] \quad j \in 1..n}{a.l_j \leftarrow \varsigma(x)b \to [l_j = \varsigma(x)b, l_i = \varsigma(x_i)b_i^{i\in(1..n)-\{i\}}]}$$

First rule claims that results are not reduced further. Second one requires to evaluate term a first then check that form to be object and then evaluate a method invocation. The last rule allows update of field only when the target has an object form. Two theorems on weak reduction is now introduced without proofs. Reader can find them in [1].

Proposition 6: Soudness of weak reduction

If $\vdash a \rightsquigarrow v$, then $a \twoheadrightarrow v$ and hence $\vdash a \leftrightarrow v$.

Theorem 7: Completness of weak reduction

Let a be a closed term and v be a result.

If $a \twoheadrightarrow v$, then there exist v' such $\vdash a \rightsquigarrow v'$.

The λ-calculus use fixpoints to represents recursive functions. The ς-calculus can express *fixpoint* operator as well. It can be encoded as object:

$$\text{fix} \triangleq [\text{arg} = \varsigma(x)x.\text{arg}, \text{val} = \varsigma(x)((x.\text{arg}).\text{arg}:=x.\text{val}).\text{val}]$$

Check that object fix satisfies fixpoint property is described in [1].

An untyped object calculi was specified throughout this chapter. The significant properties of calculus are mainly reduction relations and equation theory. These among others compose formal foundation of ς-calculi. Reduction relations determine way of reducing ς-terms and equation theory allows assuming about an equivalence of syntactical different ς-terms.

4 Type system

The first order type system for ς-calculi is declared in this section. It requires extend syntax of calculus with type terms. The rules of type system are specified by *formal syntax fragments*. Each fragment consists of a set of related rules. Each rule has premise judgments above line and one conclusion judgment bellow. Fragments may be combined to create formal system of required calculus. Here only one kind of calculus will be introduced. We choose ς-calculus with λ functions, that allows to pass argument to methods, because it is suitable for our future purpose.

Definition 8: Syntax of typed ς-calculus

A, B :: =	types	
K		ground type
$[l_i:B_i^{i\in 1..n}]$		object type (l_i distinct)
$A\rightarrow B$		function type
a, b :: =	terms	
x		variable
$[l_i=\varsigma(x_i)b_i^{i=(1..n)}]$		object (l_i distinct)
$a.l$		method invocation
$a.l \leftarrow\!\shortmid \varsigma(x)b$		method update
$\lambda(x:A)b$		function
$b(a)$		application

First formal fragment is Δ_{Ob} that has correspondence to object types. It has to be noted that two type of judgement is used. A type judgement $E\vdash B$ stating that B is well-formed type in environment E, and value typing judgement $E\vdash b:B$ stating that b has type B in environment E.

(Type Object) (l_i distinct)

$$\frac{E \rhd B_i \quad \forall i \in 1..n}{E \rhd [l_i : B_i^{i\in 1..n}]}$$

(Val Object) (where $A\equiv[l_i:B_i^{i\in 1..n}]$)

$$\frac{E, x_i : A \rhd b_i : B_i \quad \forall i \in 1..n}{E \rhd [l_i = \varsigma(x_i : A)b_i : B_i^{i\in 1..n}] : A}$$

(Val Select)

$$\frac{E \rhd a : [l_i : B_i^{i\in 1..n}] \quad j \in 1..n}{E \rhd a.l_j : B_j}$$

(Val Update) (where $A\equiv[l_i:B_i^{i\in 1..n}]$)

$$\frac{E \rhd a : A \quad E, x : A \rhd b : B_j \quad j \in 1..n}{E \rhd a.l_j \leftarrow \varsigma(x : A)b : A}$$

The first rule states that object $[l_i:B_i{}^{i\in 1..n}]$ is well-formed in environment E when each type B_i is well-formed in E. Rule (Val Object) allows to construct object of type $[l_i:B_i{}^{i\in 1..n}]$ in environment E. Object has collection of method of types B_1... B_n with self parameter of type $[l_i:B_i{}^{i\in 1..n}]$. The rule (Val Select) describes assignment of type during method invocation. Finally, rule (Val Update) deals with field update and asserts that type of object remains unchanged. Next three standard first-order fragments will be defined.
The fragment Δ_x describes how to build environments, and how to extract the type of a variable from an environment. The judgment $E\vdash\diamond$ asserts that E is well-formed. Expression $x\in E$ indicate that x is defined in E. The first rule states that empty environment is a well-formed environment. The rule (Env x) allows add a new fresh variable of existing type to an environment. The rule (Val x) extracts an assumption from an environment. The special notation E', x:A, E'' is used only to express that x:A may occur somewhere in the environment.

(Env ϕ)
$$\frac{}{\varphi \rhd \Diamond}$$

(Env x)
$$\frac{E \rhd A \quad x \notin dom(e)}{E, x:A \rhd \Diamond}$$

(Val x)
$$\frac{E', x:A, E'' \rhd \Diamond}{E', x:A, E'' \rhd x:A}$$

The special fragment Δ_K is used to introduce a ground type K. The standard use of K is as starting point for building of types.

(Type Const)
$$\frac{E \rhd \Diamond}{E \rhd K}$$

The last of introduced fragments $\Delta_\rightarrow$ deals with functions and functions types.

(Type Arrow)
$$\frac{E \rhd A \quad E \rhd B}{E \rhd A \rightarrow B}$$

(Val Fun)
$$\frac{E, x:A \rhd b:B}{E \rhd \lambda(x:A)b : A \rightarrow B}$$

(Val Appl)
$$\frac{E \rhd b: A \rightarrow B \quad E \rhd a:A}{E \rhd b(a):B}$$

By the first rule a function type $A\rightarrow B$ can be obtained from two well-formed types A and B. The second rule is used to prove that function $\lambda(x:A)b$ has type $A\rightarrow B$ from premises that telling us that b has type B under the assumption x has type A. The last rule states that a term b of function type $A\rightarrow B$ can be applied to an argument a of type A; the result has type B.

The reduction relations from untyped ς-calculus can be extended to typed terms. Here extension of the weak reduction relations is provided.

(Red Object) (where $v \equiv [l_i=\varsigma(x_i:A)b_i{}^{i\in 1..n}]$)

$$\frac{}{v \rightarrow v}$$

(Red Select) ($v' \equiv [l_i=\varsigma(x_i:A)b_i\{x_i\}^{i\in 1..n}]$)
$$\frac{a \rightarrow v' \quad b_j\{v'\} \rightarrow v \quad j \in 1..n}{a.l_j \rightarrow v}$$

(Red Update)
$$\frac{a \rightarrow [l_i = \varsigma(x_i : A)b_i^{i\in 1..n}] \quad j \in 1..n}{a.l_j \leftarrow \varsigma(x:A)b \rightarrow [l_j = \varsigma(x:A)b, l_i = \varsigma(x_i:A)b_i^{i\in(1..n)-\{i\}}]}$$

This section is closed with equation theory of typed ς-calculus. A new type of judgment is used in rules. The judgment $E \vdash b \leftrightarrow c : A$ asserts that b and c are equivalent when considered as elements of type A. The rules of fragment $\Delta_{=}$ define symmetry and transitivity and limited form of reflexivity for variables.

(Eq Symm)

$$\frac{E \rhd a \leftrightarrow b : A}{E \rhd b \leftrightarrow a : A}$$

(Eq Trans)

$$\frac{E \rhd a \leftrightarrow b \quad E \rhd b \leftrightarrow c}{E \rhd a \leftrightarrow c}$$

(Eq x)

$$\frac{E', x : A, E'' \rhd \lozenge}{E', x : A, E'' \rhd x \leftrightarrow x : A}$$

The next fragment Δ_{Obj} consists of rules for objects.

(Eq Object) (where $A \equiv [l_i : B_i^{i \in 1..n}]$)

$$\frac{E, x_i : A \rhd b_i \leftrightarrow b_i' : B_i \quad \forall i \in 1..n}{E \rhd [l_i = \varsigma(x_i) b_i : B_i^{i \in 1..n}] \leftrightarrow [l_i = \varsigma(x_i) b_i' : B_i^{i \in 1..n}]}$$

(Eq Select)

$$\frac{E \rhd a \leftrightarrow a' : [l_i : B_i^{i \in 1..n}] \quad j \in 1..n}{E \rhd a.l_j \leftrightarrow a'.l_j : B_j}$$

(Eq Update) (where $A \equiv [l_i : B_i^{i \in 1..n}]$)

$$\frac{E \rhd a \leftrightarrow a' : A \quad E, x : A \rhd b \leftrightarrow b' : B_j \quad j \in 1..n}{E \rhd a.l_j \leftarrow \varsigma(x : A) b \leftrightarrow a'.l_j \leftarrow \varsigma(x : A) b' : A}$$

(Eval Select) (where $A \equiv [l_i : B_i^{i \in 1..n}]$, $a \equiv [l_i = \varsigma(x_i : A) b_i^{i \in 1..n}]$)

$$\frac{E \rhd a : A \quad j \in 1..n}{E \rhd a.l_j \leftrightarrow b_j\{a\} : B_j}$$

(Eval Update) (where $A \equiv [l_i : B_i^{i \in 1..n}]$, $a \equiv [l_i = \varsigma(x_i : A) b_i^{i \in 1..n}]$)

$$\frac{E \rhd a : A \quad E, x : A \rhd b : B_j \quad j \in 1..n}{E \rhd a.l_j \leftarrow \varsigma(x : A) b \leftrightarrow [l_j = \varsigma(x : A) b, l_i = \varsigma(x_i : A) b_i^{i \in (1..n) - \{j\}}] : A}$$

For functions there is a standard theory for the first-order λ-calculus:

(Eq Fun)

$$\frac{E, x : A \rhd b \leftrightarrow b' : B}{E \rhd \lambda(x : A) b \leftrightarrow \lambda(x : A) b' : A \to B}$$

(Eq Appl)

$$\frac{E \rhd b \leftrightarrow b' : A \to B \quad E \rhd a \leftrightarrow a' : A}{E \rhd b(a) \leftrightarrow b'(a') : B}$$

(Eval Beta)

$$\frac{E \rhd \lambda(x : A) b\{x\} : A \to B \quad E \rhd a : A}{E \rhd (\lambda(x : A) b\{x\})(a) \leftrightarrow b\{a\} : B}$$

(Eval Beta)

$$\frac{E \rhd b : A \to B \quad x \notin dom(E)}{E \rhd \lambda(x : A) b(x) \leftrightarrow b : A \to B}$$

In this chapter the first-order type system of ς-calculi was built. It covers extent syntax of untyped calculi, definition formal fragments of object types, type environment with ground and function types. Also equation theory was revised to support types. Reduction relation of typed ς-calculi was included with small changes, but the one that appears in the typed calculi has one great merit. It assures in spite of reduction relation of untyped calculi that reducing never fails due to inconsistency.

5 Specification language and deduction rules

This section gives definition of formal object-oriented specification calculus. The calculus is intended to support creation specification on different levels of abstraction. The use of abstraction is important for specification and verification. The proposed calculus consists of specification part and set of deduction rules. It extends ς-calculus of Abadi and Cardelli. The most significant extension is introducing of quantifiers and boolean expressions into core of calculus. Rules of Gentzen system are used in a new formal system fragment that represents the deduction system of calculus. It requires incorporate the type Bool as a predefined ground type. The definition 9 introduces the Bool type. The Bool type is defined for each type that appears in the type environment. The reason is that Bool type covers conditional, which using has to satisfy type rules.

Definition 9: Bool Type

For each type A we define boolean type $Bool_A$ as follows:

$Bool_A \triangleq [if{:}A, then{:}A, else{:}A]$

$true_A : Bool_A \triangleq$

$[if = ς(x{:}Bool_A)\ x.then,\ then = ς(x{:}Bool_A)\ x.then,\ else = ς(x{:}Bool_A)\ x.else]$

$false_A : Bool_A \triangleq$

$[if = ς(x{:}Bool_A)\ x.else,\ then = ς(x{:}Bool_A)\ x.then,\ else = ς(x{:}Bool_A)\ x.else]$

The abbreviation is introduced to make notation more convenient.

$if_A\ b\ then\ c\ else\ d : A \triangleq$

$((b.then \leftarrow\!\!\!\!+ ς(x{:}Bool_A)c).else \leftarrow\!\!\!\!+ ς(x{:}Bool_A)\ d).\ if \qquad x \notin FV(c) \cup FV(d)$

Before the syntax rules for calculus will be given we declare two other notations that denotes two important kinds of terms. The first is a predicate term that is represented of λ abstraction. It must have result of type Bool. It intuitively corresponds to a predicate in the first-order calculus. Second one denotes both binding operators (quantifiers) of predicate calculus. It takes place in definition of class later.

$p(a_i^{\,i\in 1..n}) \triangleq \lambda(a_i{:}A_i)b^{\,i\in 1..n} : Bool_B$

$f(a{:}A) \triangleq \forall(x{:}A)\ a : Bool_B$ or $\exists(x{:}A)\ a : Bool_B$

The syntax of calculus is built from syntax rules of ς-calculus by adding terms consist of expressions of predicate calculus. The type terms are extended with Bool types for every defined type. Predicate definition is given explicitly although function definition covers it. The equality of ς-term operator makes possible comparing of terms inside the calculus.

Definition 10: The syntax rules of specification language

A, B :: =		types
	K	ground type
	$Bool_A$	bool type (for all non bool types A)
	$[l_i{:}B_i^{\,i\in 1..n}]$	object type (l_i distinct)
	$A \rightarrow B$	function type
a, b :: =		terms
	$x{:}A$	variable
	$[l_i{=}ς(x_i{:}A)b_i{:}B_i^{\,i=(1..n)}]{:}A$	object (l_i distinct)

(a:A).l:B	method invocation
(a:A).l:B $\mapsfrom$ ς(x:A)b:B	method update
λ(x:A)b:B	function definition
b:B(a:A)	function application
$\forall$(x:A) a :Bool_B	binding expression (universal)
$\exists$(x:A) a :Bool_B	binding expression (existencial)
a:$\text{Bool}_A \Rightarrow$ a:Bool_B	boolean expression (implication)
a:$\text{Bool}_A \wedge$ a:Bool_B	boolean expression (and)
a:$\text{Bool}_A \vee$ a:Bool_B	boolean expression (or)
$\neg$ a:Bool_A	boolean expression (not)
a:A = b:A :Bool_A	terms equality
$p(a_i{:}A_i^{i\in 1..n}){:}\text{Bool}_B$	predicate definition

Since we are developing specification language with object-oriented features every specification consists of set of classes. Definition of class for ς-calculus is introduced in [1]. We only extend every class definition with set of predicates, axioms and theorems. They make assumption on class and support abstract definitions.

Definition 11: Classes

The type of class that generating objects of type $A \equiv [l_i{:}B_i^{i\in 1..n}]$ is defined:

$$Class(A) = [new: A, l_i : A \to B_i^{i\in 1..n}, l_j : A \to B_j^{j\in n..m}, l_k = f_k : A_k^{k\in m..p}]$$

These classes have form:

$$[new=\varsigma(z{:}Class(A))[l_i=\varsigma(s{:}A)z.l_i(s)^{i\in 1..n}],\ l_i=\lambda(s{:}A)b_i^{i\in 1..n},\ l_j=p_j(a_i^{i\in 1..n})^{j\in n..m}, l_k=f_k(a_k)^{k\in m..p}]$$

Everything necessary for specification part of calculus has been defined. The notation is for the first time used to describe the trivial system – a digital clock that displays seconds, minutes and hours. The object representing clock system consists of fields that hold actual hour, minute and second. Three methods are used to increment second, minute and hour counters. Only one property what we want to examine is whether clock behaves correct. It is described with clockTickInv invariant. The example shows complete definition of class. Every counters of clock are initialized to zero when instance is created and all methods are explicitly specified. Nevertheless it is not necessary and in some cases it is more convenient to use claims on methods and deals only with significant properties of specified class.

Example 12: Clock system

```
ClockType ≙ [h:Nat, m:Nat, s:Nat, ts:Clock, tm:Clock, th:Clock]

ClockClass ≙ [
        new = ς(z: ClockType)[
                h= ς(x:ClockType)z.h(x),
                m=ς(x:ClockType)z.m(x),
                s = ς(x:ClockType)z.s(x),
                ts =ς(x:ClockType)z.ts(x),
                tm = ς(x:ClockType)z.tm(x),
                th = ς(x:ClockType)z.th(x)],
        h = λ(x:ClockType)0,
        m = λ(x:ClockType)0,
        s = λ(x:ClockType)0,
        ts = λ(x:ClockType) if x.s = 60 then x.tm else x.s ↤ (x.s + 1),
        tm = λ(x:ClockType) if x.m = 60 then x.th else (x.m ↤ (x.m+1)).s ↤ 0,
        th = λ(x:ClockType) if x.h = 24 then ((x.h ↤ 0).m ↤ 0).s ↤ 0
                                    else (x.h ↤ (x.h+1)).m ↤ 0).s ↤ 0,
        validH = λ(c:ClockType) c.h.range(0, 24)
        validM = λ(c:ClockType) c.m.range(0, 60)
        validS = λ(c:ClockType) c.s.range(0, 60)
        valid = λ(c:ClockType) validH(c) ∧ validM(c) ∧ validS(c)
        clockTickInv = ∀(c:ClockType) valid(c) ⇒ valid(c.ts)
]
```

The deduction system contains rules that are used during proving to construct proof tree. The goal is to construct a *proof tree* which is complete, in the sense that all the leaves are recognized as true. Each node of the proof tree is a *proof goal*. Each proof goal is a *sequent* consisting of sequence of formulas called *antecedents* and a sequence of formulas called *consequents*. These are known as a *sequent calculus*. Since formal system fragments use same notation of rules as sequent calculus we define it in the same manner.
At the beginning the propositional rules are given. They are quite straightforward. There is one propositional axiom that use syntactic equivalence. The Cut rule can be seen as a mechanism for introducing a case-split into a proof a sequent $\Gamma \vdash \Delta$ to yield the subgoals $\Gamma, A \vdash \Delta$ and $\Gamma \vdash A, \Delta$. It can be seen as assuming A along one branch and $\neg A$ along other.

(Prop Axiom) (a:A ≡ b:A)

$$\frac{}{\Gamma, a \rhd b, \Delta}$$

(Cut)

$$\frac{\Gamma, a \rhd \Delta \quad \Gamma \rhd a, \Delta}{\Gamma \rhd \Delta}$$

The rule for equality is based on equality theory for typed ς-calculus and is formalized using one sequent rule:
(Eq Axiom) (where a↔b:A)

$$\frac{}{\Gamma \rhd a = b, \Delta}$$

There are two rules for each of the propositional connectives of conjunction, disjunction, implication, and negation, corresponding to the antecedent and consequent occurrences of these connectives.

(Antecedent Conjunction)

$$\frac{a, b, \Gamma \rhd \Delta}{a \wedge b, \Gamma \rhd \Delta}$$

(Consequent Conjunction)

$$\frac{\Gamma \rhd a, \Delta \quad \Gamma \rhd b, \Delta}{\Gamma \rhd a \wedge b}$$

(Antecedent Disjunction)

$$\frac{a, \Gamma \rhd \Delta \quad b, \Gamma \rhd \Delta}{a \vee b, \Gamma \rhd \Delta}$$

(Consequent Disjunction)

$$\frac{\Gamma \rhd a, b, \Delta}{\Gamma \rhd a \vee b, \Delta}$$

(Antecendent Implication)

$$\frac{b, \Gamma \rhd \Delta \quad \Gamma \rhd a, \Delta}{a \Rightarrow b, \Gamma \rhd \Delta}$$

(Consequent Implication)

$$\frac{\Gamma, a \rhd b, \Delta}{\Gamma \rhd a \Rightarrow b, \Delta}$$

(Antecendent Negation)

$$\frac{\Gamma \triangleright a, \Delta}{\Gamma, \neg a \triangleright \Delta}$$

(Consequent Negation)

$$\frac{\Gamma, a \triangleright \Delta}{\Gamma \triangleright \neg a, \Delta}$$

The quantifier rules are stated bellow. The notation a{x←t} represents the result of substitution of the term for all the free occurrences of x in a with possible renaming of bound variables in a to avoid capturing any free variables in t. In the two others rules b is must be a new constant that does not occur in the conclusion sequent.

(Antecedent Universal) (where t∉FV(a))

$$\frac{\Gamma, a\{x \leftarrow t\} \triangleright \Delta}{\Gamma, \forall(x:A)a \triangleright \Delta}$$

(Consequent Universal)

$$\frac{\Gamma \triangleright a\{x \leftarrow b\}, \Delta}{\Gamma \triangleright \forall(x:A)a, \Delta}$$

(Antecedent Existential) (where b∉FV(b))

$$\frac{\Gamma, a\{x \leftarrow b\} \triangleright \Delta}{\Gamma, \exists(x:A)a \triangleright \Delta}$$

(Consequent Existential)

$$\frac{\Gamma \triangleright a\{x \leftarrow t\}, \Delta}{\Gamma \triangleright \exists(x:A)a, \Delta}$$

Very useful are rules for if. They allow to manipulate with if expression during proof and sometimes are necessary when reason of partial definitions.

(Antecedent Lift It)

$$\frac{\Gamma, \mathit{if\ a\ then\ } b\{c\} \mathit{\ else\ } b\{d\} \triangleright \Delta}{\Gamma, b\{\mathit{if\ \ a\ then\ c\ else\ d}\} \triangleright \Delta}$$

(Consequent Lift If)

$$\frac{\Gamma \triangleright \mathit{if\ a\ then\ } b\{c\} \mathit{\ else\ } b\{d\}, \Delta}{\Gamma \triangleright b\{\mathit{if\ \ a\ then\ c\ else\ d}\}, \Delta}$$

(Antecedent Lift)

$$\frac{\Gamma, a, b \triangleright \Delta \quad \Gamma, \neg a, c \triangleright \Delta}{\Gamma, \mathit{if\ \ a\ then\ b\ else\ c} \triangleright \Delta}$$

(Consequent If)

$$\frac{\Gamma, a \triangleright b, \Delta \quad \Gamma, \neg a \triangleright c\Delta}{\Gamma \triangleright \mathit{if\ \ a\ then\ b\ else\ c}, \Delta}$$

We use same example to show how the deduction system works. Proof of clockTickInv is constructed. Not all steps are presents some evident ones are omitted. Since the whole proof is quite long only the one branch examined. Nevertheless the others can be done in the same way.

Example 13: Proof of clockTickInv

⊢∀(c:Clock) : valid(c) ⇒ valid(c.ts)

⊢valid(c1) ⇒ valid(c1.ts)

valid(c1) ⊢ valid(c1.ts)

```
valid(c1)⊢valid(
        if c1.s = 60 then
                if c1.m = 60 then
                        if c1.h = 24 then ((c1.h↤0).m↤0).s↤0
                        else ((c1.h↤c1.h+1).m↤0).s↤0
                else (c1.m↤c1.m+1).s↤0
        else c1.s↤c1.s+1)
c1.h.range(0,24) ∧ c1.m.range(0,60) ∧ c1.s.range(0,60) ⊢
        (if c1.s = 60 then
                if c1.m = 60 then
                        if c1.h = 24 then ((c1.h↤0).m↤0).s↤0
                        else ((c1.h⤆c1.h+1).m⤆0).s⤆0
                  else (c1.m⤆c1.m+1).s⤆0
            else c1.s⤆c1.s+1).h.range(0,24) ∧ ...
CASE c1.s < 60 ∧ c1.m < 60 ∧ c1.h < 24:
c1.h.range(0,24) ∧ c1.m.range(0,60) ∧ c1.s.range(0,60) ⊢ (c1.s⤆c1.(s+1)).h.range(0,24)
...
```

In this chapter formal specification language and deduction system was build above ς-calculus. A set of formal system fragments was used to formalize deduction rules. The complete deduction system uses this set of rules for predicate logic together with previously defined reduction rules for the reasoning about pure ς-terms and equation theory of ς-calculus for equality reasoning. We obtain complete and soundness deduction system for suggested specification language by composition of these three parts.

6 Conclusion and future work

In this paper we built a formal specification language that covers a subset object orientation principles. Deduction system of predicate logic was adopted for reasoning about specification. Although proposed calculus is sufficient for reasoning about abstract specification of object-oriented systems many features of object orientation are missing and are primary subject of future work. It means mainly of enabling inheritance in the calculus, that brings one of the most demanded benefits – reusability of specification. The reusability is supported by another techniques – parametrized class definition. Although calculus has expressiveness to describe concurrent systems, these specifications can be quite difficult to prove. To support them the deduction system can be extended with proof strategies as it is known from others specification systems. Another way is to use a temporal logic within the calculus that could facilitate the specification of dynamic and static properties of object-based systems.

References

[1] M. Abadi and L. Cardelli. A Theory of Objects, Springer-Verlang, New York, 1996

[2] J. Martin. Principles of object-oriented analysis and design, Prentice Hall, New York, 1993

[3] L. Lamport. Specifying Systems, Addison-Wesley, Boston, 2002

[4] M. Abadi and R. M. Leino. A logic of object-oriented programs, Tapsoft'97, 1997

[5] G. Smith, An Object-Oriented Approach to Formal Specification, PhD Thesis, University of Queensland, 1992

[6] N. Shankar, S.Owre, J.M. Rushby, and D.W. Stringer-Calvert. PVS Prover Guide. Computer Science Laboratory, SRI International, Menlo Park, CA, 1999

[7] V. Cordeiro, A.Sampaio, S. Meira. From MooZ to Eiffel – A Rigorous Approach to System Development, FME'94, Springer-Verlang, 1994

[8] J. Zlatuska. λ-calculus, Grafex Blansko, Brno, 1993