Testing object Interactions

Grüner, A.

Chapter 2

Java-like sequential programming language – *Japl*

This chapter introduces a *Java*-like programming language which we will use for further investigations. The intention is to provide a language that, on the one hand, captures a reasonable subset of features many modern object-oriented general-purpose programming languages like *Java* and C♯ have in common and that, on the other hand, comes with a formal semantics which allows to reason about the language.

Certainly there exist a couple of formal languages already aiming at *Java* or *Java*-like languages. For instance, in [1] Abadi and Cardelli suggested a core calculus for object-oriented languages. In addition they provide several extensions and modifications that deal with certain language features. Their typed imperative object calculus impς has been extended by Gordon and Hankin with concurrency [31] and a modification of the concurrent object calculus in turn was extended with classes in [5, 4], and in particular in [64]. The above mentioned calculi can be considered as an object-oriented counter-part to the family of λ-calculi. In a very concise way, they capture certain general features that almost all object-oriented languages have in common. Although these approaches represent a very good basis for investigating object-oriented languages in general, the (intended) generalization has its price. For, the provided abstract syntax is quite different from *Java* or C♯. Hence, it is sometimes not easy to find a *Java* program that corresponds to a given program of one of these object calculi and vice versa. Moreover, some language features are considered as special cases of other features. For example, in [1] there is no distinction between fields and methods which means that not only fields but also methods can be updated. Again, aiming at object-oriented languages in general, this represents an elegant unification. Since we restrict our approach to C♯-like and *Java*-like languages, however, this kind of

design decisions entails unnecessary complications.

However, there exist other approaches for capturing object-oriented languages which are closer to *Java*. Two prominent examples are *Featherweight Java* (*FJ*) [34] and *Middleweight Java* (*MJ*) [15]. The original *FJ* does not deal so much with *Java*'s operational aspects but rather captures *Java*'s type system. Thus, it is mainly used for investigating subtyping, inheritance, generics, and the like. *MJ*, in contrast, can be seen as an extension of *FJ* with respect to many of *Java*'s operational features most notably *MJ* introduces many imperative features.

The language that we propose here lies somewhere between *FJ* and *MJ*. In particular, our language is class-based, i.e., a program basically consists of a set of class definitions from which objects can be instantiated at runtime. Each object comprises a set of fields (also known as instance variables) and a set of methods. Objects are referenced by names which can be passed around giving rise to aliasing. The language is imperative, fields and variables allow for destructive updates. Furthermore, recursive method calls are possible.

To simplify matters, we do not consider, however, subtyping and inheritance. We will discuss these features and other possible extensions of the programming language in Chapter 5. We also omit more specific concepts like interface definitions, anonymous classes, generics, delegations, and reflection. Furthermore, in this part we focus on a sequential setting, that is, the language only allows for a single-threaded flow of control.

The rest of this chapter is structured as follows. In the first three sections, we will present the syntax, the type system, and, respectively, the operational semantics of closed programs of our language. A *closed program* is a self-contained entity in the sense that its possible behavior is completely determined by the given program code. In Section 2.4, in contrast, we will extend the language with the notion of *components* which will allow programs to contain references to classes that are not defined within the program code but are assumed to be provided by the program's environment. Finally we will conclude this chapter by presenting our testing approach in context of the new language and compare it with traditional unit testing.

2.1 Syntax

The grammar of the *Java*-like programming language is given in Table 2.1. A program consists of a list of global variables, a set of classes, and a main program (or main body). Note, that due to simplicity, our language slightly differs from *Java* already on the program level in two aspects: first, *Java* does not provide a designated construct for specifying global variables but rather requires them to be introduced by static fields. Second, in *Java* also the main program is not represented by a special construct on the program level but is given by a static method with a special name. However, to keep the language small and simple, we omit static fields and methods. Adding special constructs also allows for a clearer separation of concerns.

p	$::= \overline{T}\,\overline{x};\ \overline{cldef}\ \{stmt;\ \texttt{return}\}$	program
$cldef$	$::= \texttt{class}\ C\{\overline{T}\,\overline{f};\ con\ \overline{mdef}\}$	class definition
con	$::= C(\overline{T}\,\overline{x})\{\overline{T}\,\overline{x};\ stmt;\ \texttt{return}\}$	constructor
$mdef$	$::= T\ m(\overline{T}\,\overline{x})\{\overline{T}\,\overline{x};\ stmt;\ \texttt{return}\ e\}$	meth. definition
$stmt$	$::= x = e \mid x = e.m(e,\ldots,e) \mid x = \texttt{new}\ C(e,\ldots,e)$	statements
	$\mid f = e \mid \varepsilon \mid stmt; stmt \mid \{\overline{T}\,\overline{x};\ stmt\}$	
	$\mid \texttt{while}\ (e)\ \{stmt\} \mid \texttt{if}\ (e)\ \{stmt\}\ \texttt{else}\ \{stmt\}$	
e	$::= x \mid f \mid \texttt{null} \mid \texttt{this} \mid \texttt{op}(e,\ldots,e)$	expressions

Table 2.1: Simple *Java*-like language: syntax

The language is strongly typed, in the sense that the definitions of variables, formal parameters, and methods always include a type. The set of possible types is denoted by T. The details about the type system will be given in the next section. Here it suffices to say that T comprises class names and additionally some base types like Booleans and integer, if necessary.

We assume a number of meta-variables: $C, D, \ldots$ range over the set of class names *CNames*; $x, y, \ldots$ range over the set of variables *VNames*; f ranges over the set of field names *FNames*; and m ranges over the set of method names *MNames*. To keep the definitions compact, we use $\overline{C}$ (or, respectively, $\overline{f}$ or $\overline{cldef}, \ldots$), for the, possibly empty, sequence $C_1\ \ldots\ C_n$. Similarly we use $\overline{e}$ for the comma-separated sequence $e_1, \ldots, e_n$ and $\overline{T}\,\overline{x}$; to abbreviate the sequence $T_1\,x_1; \ldots; T_n\,x_n$. In slight abuse of the sequence notation, we sometimes also use it within function applications (or judgments) to denote the sequence or the set which results from applying the function on each element of the original sequence.

A class is given by its name, its field declarations, exactly one constructor, and a set of method definitions. Again, for simplicity we do not deal with subtyping or inheritance here. We assume, furthermore, that all fields are private, i.e., every object can directly access its own fields, only.[1]

Like in *Java*, both constructor and method definitions provide a list of formal parameters as well as a body which in turn may introduce new local variables and which ends with a return term. The return term of a method always includes a return value (possibly the undefined reference `null`) whereas a constructor's return term never does. Consequently, the method definition is not only equipped with a name but also with a return type. Note, that, as in *Java*, the name of the constructor is always the name of the class itself.

A statement is either an assignment, the empty statement (denoted by ε), a sequential composition, a block statement, a while-loop, or a conditional statement. Only expressions can be assigned to fields. Variables can additionally be updated by the result of a method or a constructor call.

An expression, finally, is either a variable, a field, the undefined reference `null`,

[1] Note that *Java*'s accessibility modifier `private` has a slightly different meaning.

the self reference `this` or a built-in operation. We do not deal with the details of the built-in operations but we only assume them to exist and to be side-effect free. Furthermore, we consider constants to be built-in operations with arity zero.

Remark 2.1.1 (Set notation)**:** Although we write classes, fields, and method definitions in a sequential way, their order has no meaning. Thus, we treat some constructs rather as sets. In particular, we will sometimes use set operations like $mdef \in \overline{mdef}$ or $\overline{fdef}_1 \cup \overline{fdef}_2$ or even $cldef \in p$.

We conclude this section with a small example program written in our language. The program is given in Listing 2.1. It consists of a global variable, two class definitions for binary trees, and a short main program. The first class `Data` is used to represent some data. The second class `BinTree` represents the binary tree structure. The main program first creates a data and a tree node object. The data is stored in a locally and the tree node instance in the globally defined variable. Afterwards another data and another tree node object are created, where the first tree node is passed to the constructor of the second tree node building the left branch of the new tree node.

2.2 Static semantics

Our programming language is statically typed. Thus, well-formedness of a program implies that we can associate a type with each of the program's variables and names such that the type assignments are consistent with regards to certain typing rules. We use type mappings, Γ and Δ, to denote these type assignments; for instance $\Gamma(x)$ either yields the type associated to variable x or is undefined. The mapping Γ provides the typing information of names which only have a local scope like variables and fields. It has a stack structure: appending a typed variable $x{:}T$ to an existing local mapping Γ, separated by a comma, creates a new mapping equal to Γ but extended by the new x which might shadow a possibly existing x in Γ.

In contrast, the mapping Δ contains the typing information of globally accessible constructs, namely of classes. Also for global mappings, we express its extension by appending typed names. However, the global mapping is not stack structured, since all names of classes are assumed to be different, hence, we don't have to deal with shadowing.

To express well-typedness, we introduce two kinds of typing judgments. Well-typedness of an expression e is denoted by a judgment of the following form:

$$\Gamma; \Delta \vdash e : T \ .$$

More precisely, the judgment states that, under the assumption of some type assignments given by $\Gamma; \Delta$, the expression e is well-typed and, additionally, that e itself is of type T. In this regard, the pair of type mappings $\Gamma; \Delta$ represents an assumption about the typed names provided by the environment of the expression e and we will refer to it as a *typing context*. The second kind of typing judgments

Listing 2.1: Simple example: Binary tree

```
BinTree s;

class Data {
  Data() { return }
}

class BinTree {
  BinTree lbranch;
  BinTree rbranch;
  Data value;

  BinTree(Data v, BinTree l, BinTree r) {
    value = v;
    lbranch = l;  rbranch = r;
    return
  }

  BinTree getLeft() { return lbranch; }

  BinTree getRight() { return rbranch; }

  Data getData() { return value; }

  BinTree setData(Data v) {
    value = v;
    return this;
  }
}

{
  { Data v;
    v = new Data();
    s = new BinTree(v, null, null);
    v = new Data();
    s = new BinTree(v, s, null)
  };
  return
}
```

is used for expressing well-typedness of entire programs and its syntactical constituents up to statements. Well-typedness of such a code fragment s is denoted by a judgment of the following form:

$$\Gamma; \Delta \vdash s : \mathsf{ok} \ .$$

Note that, in contrast to expressions, s does not provide a type.

Finally, we use the typing judgments to formalize the typing rules of our language. We provide the typing rules in the form of inference rules where each

$$[\text{T-Prog}]\ \frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T} \qquad \Delta' = \Delta, \mathit{cltype}(\overline{\mathit{cldef}}) \qquad \Gamma'; \Delta' \vdash \overline{\mathit{cldef}} : \mathsf{ok} \qquad \Gamma'; \Delta' \vdash \mathit{stmt} : \mathsf{ok}}{\Gamma; \Delta \vdash \overline{T}\ \overline{x};\ \overline{\mathit{cldef}}\ \{\mathit{stmt};\ \mathtt{return}\} : \mathsf{ok}}$$

$$[\text{T-Class}]\ \frac{\Gamma' = \Gamma, \overline{f}{:}\overline{T}, \mathtt{this}{:}C \qquad \Gamma'; \Delta \vdash \mathit{con} : \mathsf{ok} \qquad \Gamma'; \Delta \vdash \overline{\mathit{mdef}} : \mathsf{ok}}{\Gamma; \Delta \vdash \mathtt{class}\ C\{\overline{T}\ \overline{f};\ \mathit{con}\ \overline{\mathit{mdef}}\} : \mathsf{ok}}$$

$$[\text{T-Con}]\ \frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T}, \overline{x'}{:}\overline{T'} \qquad \Gamma'; \Delta \vdash \mathit{stmt} : \mathsf{ok}}{\Gamma; \Delta \vdash C\ (\overline{T}\ \overline{x})\{\overline{T'}\ \overline{x'};\ \mathit{stmt};\ \mathtt{return}\} : \mathsf{ok}}$$

$$[\text{T-MDef}]\ \frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T}, \overline{x'}{:}\overline{T'} \qquad \Gamma'; \Delta \vdash \mathit{stmt} : \mathsf{ok} \qquad \Gamma'; \Delta \vdash e : T}{\Gamma; \Delta \vdash T\ m(\overline{T}\ \overline{x})\{\overline{T'}\ \overline{x'};\ \mathit{stmt};\ \mathtt{return}\ e\} : \mathsf{ok}}$$

$$[\text{T-VUpd}]\ \frac{\Gamma; \Delta \vdash e : \Gamma(x)}{\Gamma; \Delta \vdash x = e : \mathsf{ok}} \qquad [\text{T-FUpd}]\ \frac{\Gamma; \Delta \vdash e : \Gamma(f)}{\Gamma; \Delta \vdash f = e : \mathsf{ok}}$$

$$[\text{T-Call}]\ \frac{\Gamma; \Delta \vdash e : C \quad \Gamma; \Delta \vdash \overline{e} : \Delta(C)(m).\mathit{dom} \quad \Gamma; \Delta \vdash x : \Delta(C)(m).\mathit{ran}}{\Gamma; \Delta \vdash x = e.m(\overline{e}) : \mathsf{ok}}$$

$$[\text{T-New}]\ \frac{\Gamma; \Delta \vdash x : C \quad \Gamma; \Delta \vdash \overline{e} : \Delta(C)(m).\mathit{dom}}{\Gamma; \Delta \vdash \mathtt{new}\ x = C(\overline{e}) : \mathsf{ok}}$$

$$[\text{T-Seq}]\ \frac{\Gamma; \Delta \vdash \mathit{stmt}_1 : \mathsf{ok} \quad \Gamma; \Delta \vdash \mathit{stmt}_2 : \mathsf{ok}}{\Gamma; \Delta \vdash \mathit{stmt}_1;\ \mathit{stmt}_2 : \mathsf{ok}}$$

$$[\text{T-Block}]\ \frac{\Gamma, \overline{x}{:}\overline{T}; \Delta \vdash \mathit{stmt} : \mathsf{ok}}{\Gamma; \Delta \vdash \{\overline{T}\ \overline{x};\ \mathit{stmt}\} : \mathsf{ok}}$$

$$[\text{T-While}]\ \frac{\Gamma; \Delta \vdash e : \mathsf{bool} \qquad \Gamma; \Delta \vdash \mathit{stmt} : \mathsf{ok}}{\Gamma; \Delta \vdash \mathtt{while}\ (e)\ \{\mathit{stmt}\} : \mathsf{ok}}$$

$$[\text{T-Cond}]\ \frac{\Gamma; \Delta \vdash e : \mathsf{bool} \qquad \Gamma; \Delta \vdash \mathit{stmt}_1 : \mathsf{ok} \qquad \Gamma; \Delta \vdash \mathit{stmt}_2 : \mathsf{ok}}{\Gamma; \Delta \vdash \mathtt{if}\ (e)\ \{\mathit{stmt}_1\}\ \mathtt{else}\ \{\mathit{stmt}_2\} : \mathsf{ok}}$$

Table 2.2: Simple *Java*-like language: type system (program parts up to stmts)

rule's conclusion consists of a specific judgment. If an instance of a typing rule is derivable then the corresponding code fragment in the conclusion is well-typed.

Before we introduce the typing rules, we make up for the missing definition of the types T.

Definition 2.2.1 (Types)**:** The set of types T of the programming language is given by means of the following grammar:

$$T ::= U \mid (\mathit{MNames} \cup \mathit{CNames}) \rightharpoonup (U \times \ldots \times U \to U)$$
$$U ::= C \mid \texttt{bool} \mid \mathbf{B}$$

Thus, the set of types comprises class names, class types, a Boolean type and, if necessary , some additional base types $\mathbf{B}$. Class names are used as types for objects whereas classes are typed with regards to their provided interface: a class type T is a partial function that maps each of the class' method and constructor name to a pair consisting of the parameter types and the return type. For methods m of the corresponding class, we will use $T(m).\mathit{ran}$ and $T(m).\mathit{dom}$ to denote the projection onto the first and, respectively, onto the second element of the pair, i.e., on the method's parameter types and, correspondingly, its return type. We use the same notation for the constructor name C of the class, where $T(C).\mathit{ran}$ always equals C.

Let us now discuss the typing rules in detail. Table 2.2 deals with the typing rules for programs and their syntactical constituents up to statements. Rule T-Prog stipulates that a program is well-typed if its class definitions and its main statement are well-typed. The set of premises representing the type checks of the class definitions is subsumed by using the sequence notation. The assumed typing contexts of both, the class definitions and the main statement, are enriched by the typed global variables and classes. Note, in order to carry out the class definitions' type-checks it is necessary to extend the type context by all class types already, as the method bodies might contain references to program classes. To this end, the type of a class is determined by the auxiliary function $\mathit{cltype}(\mathit{cldef})$ which extracts the type from the class definition's signature. It is defined as follows:

$$\mathit{cltype}(\ \texttt{class}\ C\{\ \overline{T}\ \overline{f};\ \mathit{con}\ \overline{\mathit{mdef}}\ \}\ ) \stackrel{\text{def}}{=} (C{:}f_C)\ \text{, with}$$
$$f_C : (\mathit{MNames} \cup \mathit{CNames}) \rightharpoonup (U \times \ldots \times U \to U);$$
$$n \mapsto \begin{cases} (\overline{T}, C) & \text{if } n = C \text{ and } C(\overline{T}\ \overline{x})\{\overline{T'}\ \overline{x'};\ \mathit{stmt};\ \texttt{return}\} \in \overline{\mathit{mdef}} \\ (\overline{T}, T) & \text{if } n = m \text{ and } T\ m(\overline{T}\ \overline{x})\{\overline{T'}\ \overline{x'};\ \mathit{stmt};\ \texttt{return}\ e\} \in \overline{\mathit{mdef}} \end{cases}$$

We assume that all method names of a class are distinct and that no method has the name of its class. This ensures that the function *cltype* is well-defined.

Rule T-Class deals with well-typedness of a class. A class is well-typed if its constructor and method definitions are well-typed. The type context of the constructor and method type-checks are enriched by the fields defined within the

$$[\text{T-Var}]\frac{\Gamma(x) = T}{\Gamma;\Delta \vdash x : T} \qquad [\text{T-Field}]\frac{\Gamma(f) = T}{\Gamma;\Delta \vdash f : T}$$

$$[\text{T-Null}]\ \Gamma;\Delta \vdash \texttt{null} : C \qquad [\text{T-This}]\frac{\Gamma(\texttt{this}) = C}{\Gamma;\Delta \vdash \texttt{this} : C}$$

$$[\text{T-Op}]\frac{\Gamma;\Delta \vdash \overline{e} : dom(\Delta(\texttt{op})) \quad ran(\Delta(\texttt{op})) = T}{\Gamma;\Delta \vdash \texttt{op}(\overline{e}) : T}$$

Table 2.3: Simple *Java*-like language: type system (exprs)

class as well as by the special name `this`, typed by the corresponding class, since `this` can be used for self references within the constructor and method bodies.

According to the rules T-Con and T-MDef, a constructor as well as a method definition is well-typed if the body statement is well-typed assuming a type context that is extended by the local variables and the formal parameters. For methods additionally the type of the returned expression is checked.

Regarding all the variants of assignments, we have to check that the left-hand side and the right-hand side of the equals sign are of the same type. As for method and constructor calls we additionally have to check the types of the actual parameters. The parameter checks are expressed in slight abuse of the sequence notation: the elements of the parameter sequence and the elements of the method's parameter type tuple are matched concerning their order, such that each pair gives rise to a corresponding typing judgment. In particular, both sequences have to be of the same length. The typing rule for constructor calls (T-New) stipulates that objects are typed by the name of their class.

A sequence of statements is well-typed if each sub-statement is well-typed. Similarly, a block statement is well-typed if its body statement is well-typed where the local typing context is extended by the new variables declared by the block statement.

While statements and conditional statements are well-typed if their sub-statements are well-typed and if their conditional expressions are Boolean expressions.

Table 2.3 deals with the typing of expressions. Types of variables, fields and `this` can be directly looked up in the local type context Γ. The empty reference `null` is of any class name type. As for the built-in operations, we assume a typing to be already included in the global context Δ. Applications of these operations are only well-typed if the actual parameters conform to the domain type of the operation. If so, the application is of the range type of the operation.

Definition 2.2.2 (Well-typedness)**:** A program p is *well-typed* if there exist a type mapping Δ such that the judgment

$$;\Delta \vdash p : \mathsf{ok}$$

is derivable by means of the deduction rules given in Table 2.2 and 2.3. In particular,

the deduction starts with an empty local type mapping. Therefore, well-typedness of a program p regarding a certain type context Δ is denoted by

$$\Delta \vdash p : \mathsf{ok} .$$

2.3 Operational semantics

Operational semantics [59] is a way to express the meaning of a programming language: for each language construct, the effect of its execution on an abstract machine is formalized. The operational semantics of our language will be given in form of a small-step semantics. This kind of semantics is based on the idea that a program execution is considered as a sequence of indivisible steps that manifest themselves in form of changes in the program's configuration. The small-step semantics stipulates what kind of changes may happen in a certain situation. It is often represented by a transition relation which in turn is described by an inference system where the conclusion of each inference rule determines a (parameterized) transition between two configurations. The concept of using an inference system to describe the computation step, also called structural operational semantics, goes back to Plotkin [56]. Before we take a closer look at the operational semantics' transition rules let us first discuss the constituents of a program configuration. A program configuration $(h, \mathsf{v}, \mathsf{CS})$ is a triple consisting of the current state of the heap h, the global variables v, and the call stack CS. The details about the elements of a program configuration are given in the following definition.

Definition 2.3.1 (Configuration)**:** Let the set of all possible values be denoted by Val including null as the semantical representation for `null`. We use partial functions from field names to values to represent the state of an object. More specifically, an object consists of the value of its fields and a reference to its class. Thus, we define

$$Obj \stackrel{\text{def}}{=} CNames \times \mathsf{F} \quad \text{with } \mathsf{F} \stackrel{\text{def}}{=} FNames \rightharpoonup Val$$

as the set of all possible objects. For an object $o \in Obj$ we use $o.class$ and $o.fields$ to denote the projection onto the first or, respectively, the second element of the pair.

Let N be the set of *object names*. The heap is represented by a partial function from object names to objects. We use

$$\mathsf{H} \stackrel{\text{def}}{=} N \rightharpoonup Obj$$

to denote the set of heaps.

Let $\mathsf{V} \stackrel{\text{def}}{=} VNames \rightharpoonup Val$ be the set of *variable functions*, i.e., partial functions from variables to values. The state of a program's global variables is represented by an element v of V.

The *call stack* consists of a list of *activation records* each capturing the local variables as well as the code fragment of a method instance that still has to be executed. More precisely, an activation record's code fragment is of the form:

$$mc ::= stmt^x;\ mc \mid \texttt{return}\ [e],$$

where the square brackets denote optional terms. The non-terminal $stmt^x$ represents an extension of the non-terminal $stmt$ given in Table 2.1 in that it includes a new auxiliary statement BE which is needed for a proper processing of block statements. The details will be explained later. If a method is about to return the control back to the method's callee then the corresponding method fragment of the activation record consists of the return term only. Moreover, the very first activation record might represent the execution of the main body, which explains the square brackets around the return expression (as the main body does not return a value).

The state of the local variables of a method instance is given by a list of variable functions μ. Each variable function v of that list represents the state of the variables of a single block statement. Additionally, the local variable function list of a method instance always includes a variable function for the method's parameters, which is the empty function $\mathsf{v}_\perp$ if the method does not provide any parameter:

$$\mu ::= \mathsf{v}{\cdot}\mu \mid \epsilon.$$

For the time being, we can distinguish two kinds of activation records. The top-most activation record of a call stack represents the method instance which is currently *active*, i.e., which is in fact currently in execution. These activation records are always of the form:

$$\mathsf{AR}^a ::= (\mu, mc).$$

All other activation records within a call stack represent method or constructor instances which haven't finished their execution but which have called another method or constructor that hasn't returned yet. Thus, the calling method instance is *blocked* waiting for the return value of the called method or constructor. These activation records carry an auxiliary statement in front of the actual code fragment:

$$\mathsf{AR}^b ::= (\mu, \texttt{rcv}\ x; mc).$$

The receive statement is accompanied by the variable that is to be updated by the return value of the called method or constructor.

Finally, we can give a definition for the call stack:

$$\begin{array}{rcl} \mathsf{CS}^a & ::= & \mathsf{AR}^a \circ \mathsf{CS}^b \\ \mathsf{CS}^b & ::= & \mathsf{AR}^b \circ \mathsf{CS}^b \mid \epsilon \\ \mathsf{CS} & ::= & \mathsf{CS}^a \end{array}$$

The set $Conf$ is the set of all configurations, i.e.

$$Conf \stackrel{\text{def}}{=} \mathsf{H} \times \mathsf{V} \times \mathsf{CS}.$$

Before we discuss the transition rules of the operational semantics we introduce some auxiliary functions. The first group of auxiliary functions deals with evaluating and updating the variables of a program. The definitions are given in Table 2.4. The function $eval_m$ looks up the value of a variable accessible within

$$
\begin{aligned}
eval_m(\mathsf{v} \cdot \mu, x) &\stackrel{\text{def}}{=} \begin{cases} \mathsf{v}(x) & \text{if } x \in dom(\mathsf{v}) \\ eval_m(\mu, x) & \text{otherwise} \end{cases} \\
eval(\mathsf{v}, \mu, x) &\stackrel{\text{def}}{=} eval_m(\mu \cdot \mathsf{v}, x) \\
vupd_m(\mathsf{v} \cdot \mu, x \mapsto v) &\stackrel{\text{def}}{=} \begin{cases} \mathsf{v}[x \mapsto v] \cdot \mu & \text{if } x \in dom(\mathsf{v}) \\ \mathsf{v} \cdot vupd_m(\mu, x \mapsto v) & \text{otherwise} \end{cases} \\
vupd(\mathsf{v}, \mu, x \mapsto v) &\stackrel{\text{def}}{=} (\mathsf{v}', \mu') \\
&\quad \text{where } \mu' \cdot \mathsf{v}' = vupd_m(\mu \cdot \mathsf{v}, x \mapsto v)
\end{aligned}
$$

Table 2.4: Variable evaluation

a method instance. Following the structure of the nested scopes of the local variables, $eval_m$ recursively walks through the list of variable functions and returns the first defined value of the variable. The function *eval* evaluates a variable in the context of, both, a global variable function and the local variable function list. To this end, it appends the global variable function to the local variable function list, and passes the result as input parameter to $eval_m$. This way, the local variable context is extended by the global variables while respecting the possible shadowing effect by some local variables.

Similarly, we introduce two functions for updating the variable state. The first function $vupd_m$ takes a local variable function list as well as a variable-value pair and updates the first variable function within the list which defines a value for this variable. The second function *vupd* updates a variable of a program by, again, extending the local variable function list with the global variables and applying $vupd_m$. Then it takes the result and separates the global variables from the local variable list again, in order to return the updated variable pair.

The variable evaluation functions are used in the definition of the semantics

$$
\begin{aligned}
[\![x]\!]_h^{\mathsf{v},\mu} &\stackrel{\text{def}}{=} eval(\mathsf{v}, \mu, x) \\
[\![\texttt{this}]\!]_h^{\mathsf{v},\mu} &\stackrel{\text{def}}{=} eval(\mathsf{v}, \mu, \texttt{this}) \\
[\![f]\!]_h^{\mathsf{v},\mu} &\stackrel{\text{def}}{=} \mathsf{F}(f) \text{ with } (C, \mathsf{F}) = h(eval(\mathsf{v}, \mu, \texttt{this})) \\
[\![\texttt{null}]\!]_h^{\mathsf{v},\mu} &\stackrel{\text{def}}{=} \mathsf{null} \\
[\![\mathsf{op}(\overline{e})]\!]_h^{\mathsf{v},\mu} &\stackrel{\text{def}}{=} \mathsf{op}([\![\overline{e}]\!]_h^{\mathsf{v},\mu})
\end{aligned}
$$

Table 2.5: Expression evaluation

of expressions in Table 2.5. For each expression the semantics either is undefined or yields an element of *Val* depending on a given heap h and variable context represented by the global variable function v and a local variable function list μ. The evaluation of a variable and the self reference `this` is realized by just applying the aforementioned evaluation function. A field is evaluated by looking up the value of the self reference `this` which, in turn, is used to get the corresponding object; then the object's field function yields the desired value. The keyword `null` and the built-in operations are evaluated to their semantic representations. Note that the semantical representation null of the keyword `null` must not be an object name, i.e., we require $\mathsf{null} \notin N$.

The last group of auxiliary notations, given in Table 2.6, are simple functions

$$
\begin{array}{rl}
\mathit{classes}_p \stackrel{\text{def}}{=} & \{C_1, \ldots, C_k\} \\
& \text{where} \\
& p = \overline{T}\,\overline{x};\ \texttt{class}\ C_1\{\ldots\}\ \ldots\ \texttt{class}\ C_k\{\ldots\}\ \{\mathit{stmt};\ \texttt{return}\} \\
\\
\mathit{fields}_p(C) \stackrel{\text{def}}{=} & \overline{T}\,\overline{f} \\
& \text{where } \texttt{class}\ C\{\overline{T}\,\overline{f};\ \mathit{con}\ \overline{\mathit{mdef}}\} \in p \\
\\
\mathit{cbody}_p(C) \stackrel{\text{def}}{=} & \mathit{stmt}; \\
\mathit{cparams}_p(C) \stackrel{\text{def}}{=} & \overline{T}\,\overline{x} \\
\mathit{cvars}_p(C) \stackrel{\text{def}}{=} & \overline{T'}\,\overline{x'} \\
& \text{where } \texttt{class}\ C\{\overline{T}\ \overline{f};\ \mathit{con}\ \overline{\mathit{mdef}}\} \in p \\
& \text{and } \mathit{con} = C(\overline{T}\,\overline{x})\{\overline{T'}\,\overline{x'};\ \mathit{stmt};\ \texttt{return}\} \\
\\
\mathit{mbody}_p(C,m) \stackrel{\text{def}}{=} & \mathit{stmt};\ \texttt{return}\ e \\
\mathit{mparams}_p(C,m) \stackrel{\text{def}}{=} & \overline{T}\,\overline{x} \\
\mathit{mvars}_p(C,m) \stackrel{\text{def}}{=} & \overline{T'}\,\overline{x'} \\
& \text{where } \texttt{class}\ C\{\overline{T}\ \overline{f};\ \mathit{con}\ \overline{\mathit{mdef}}\} \in p \\
& \text{and } C'\ m(\overline{T}\,\overline{x})\{\overline{T'}\,\overline{x'};\ \mathit{stmt};\ \texttt{return}\ e\} \in \overline{\mathit{mdef}} \\
\\
\mathit{Obj}^C_{p\perp} \stackrel{\text{def}}{=} & (C, \mathsf{F}) \text{ with } \mathit{dom}(\mathsf{F}) \stackrel{\text{def}}{=} \{f_1, \ldots, f_k\} \text{ and} \\
& \qquad\qquad \mathsf{F}(f_i) \stackrel{\text{def}}{=} \mathit{ival}(T_i) \text{ for all}, 1 \le i \le k \\
& \text{where } T_1\ f_1;\ \ldots;\ T_k\ f_k = \mathit{fields}_p(C)
\end{array}
$$

Table 2.6: Auxiliary notations

that extract certain syntactical fragments of a given program. They are used in the definition of the operational semantics to keep the notation clear and concise. All functions have in common that they expect a well-formed program p as argument written as an index of each function. The function *classes* yields the set of class names defined in the program. Similarly, the functions *fields* yields the sequence of field names of a given class along with their types. The functions *cbody*, *cparams*, and *cvars* return the body of a class' constructor, its parameters, and its local variables, respectively. The functions *mbody*, *mparams*, and *mvars* do the same for methods. Note that the body of a method but not the body of a constructor includes the return term. The function $Obj^C_{p\perp}$ returns an object of class C where all the fields declared within C are set to the initial value of the corresponding type. We assume that for each base type T of our language there exists a certain constant, $ival(T)$, of type T which represents the initial value for variables of that type. In particular, we assume $ival(\mathsf{bool}) \stackrel{\text{def}}{=} \mathsf{false}$, $ival(\mathsf{int}) \stackrel{\text{def}}{=} 0$, and $ival(C) \stackrel{\text{def}}{=} \mathsf{null}$ for all class names C. Whenever we will use these auxiliary functions in the following, the considered program p will always be clear from the context, such that we will leave out the index notation.

The transition rules of the operational semantics are given in Table 2.7. Most of the rules share the same pattern. The leftmost statement of the topmost activation record is reduced which might cause a change of the heap and/or the values of the variables. Some of the rules can only be applied under certain conditions, represented by corresponding premises. The rule Ass deals with the assignment to a variable by merely updating the variable state. The rule FUpd updates a field. To this end, it looks up the object name currently stored in `this`. The name is used to get the corresponding object in the heap and to update its field function. The updated object in turn is used to update the heap. For functions f we use the notation $f[x \mapsto y]$ to denote a new function f' which is identical to f for all $z \in dom(f) \setminus \{x\}$ but which additionally maps x to y. Note, that this means either an extension of the original domain of f by the new element x or a modification of the image of x.

The rule Call extends the call stack by a new activation record consisting of the method body of the called method as well as of a new variable function. The variable function assigns the callee object name to `this`, the actual parameters to the formal parameters and it initializes the local variables. Moreover, the rule adds an auxiliary statement `rcv` x to the activation record of the calling method. After returning from the called method, this statement determines the variable which is to be updated by the return value. Note, that our language does not support the notion of exceptions. Thus, a method call whose callee expression evaluates to null gets stuck, as null is never part of the domain of the heap.

Processing a constructor call resembles very much the method call, but we have to add a new initial object to the heap and associate it to an object name which is not already in use. Moreover, we do not only copy the constructor body to the new activation record but we also add a return term with a return expression that yields the new object.

$$[\textsc{Ass}]\ \frac{(\mathsf{v}', \mu') = vupd(\mathsf{v}, \mu, x \mapsto [\![e]\!]_h^{\mathsf{v},\mu})}{(h, \mathsf{v}, (\mu, x = e; mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}', (\mu', mc) \circ \mathsf{CS}^b)}$$

$$[\textsc{FUpd}]\ \frac{o = [\![\texttt{this}]\!]_h^{\mathsf{v},\mu} \quad (C, \mathsf{F}) = h(o) \quad h' = h[o \mapsto (C, \mathsf{F}[f \mapsto [\![e]\!]_h^{\mathsf{v},\mu}])]}{(h, \mathsf{v}, (\mu, f = e; mc) \circ \mathsf{CS}^b) \rightsquigarrow (h', \mathsf{v}, (\mu, mc) \circ \mathsf{CS}^b)}$$

$$[\textsc{Call}]\ \frac{\begin{array}{c} o = [\![e]\!]_h^{\mathsf{v},\mu} \quad C = h(o).class \quad \overline{T}\ \overline{x} = mparams(C, m) \quad \overline{T_l}\ \overline{x_l} = mvars(C, m) \\ \mathsf{v}_l = \{\texttt{this} \mapsto o, \overline{x} \mapsto [\![\overline{e}]\!]_h^{\mathsf{v},\mu}, \overline{x_l} \mapsto ival(\overline{T_l})\} \end{array}}{(h, \mathsf{v}, (\mu, x = e.m(\overline{e}); mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}, (\mathsf{v}_l, mbody(C, m)) \circ (\mu, \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}^b)}$$

$$[\textsc{New}]\ \frac{\begin{array}{c} o \in N \setminus dom(h) \quad h' = h[o \mapsto Obj_{\perp}^{C}] \quad \overline{T}\ \overline{x} = cparams(C) \quad \overline{T_l}\ \overline{x_l} = cvars(C) \\ \mathsf{v}_l = \{\texttt{this} \mapsto o, \overline{x} \mapsto [\![\overline{e}]\!]_h^{\mathsf{v},\mu}, \overline{x_l} \mapsto ival(\overline{T_l})\} \end{array}}{\begin{array}{l} (h, \mathsf{v}, (\mu, x = \texttt{new}\ C(\overline{e}); mc) \circ \mathsf{CS}^b) \rightsquigarrow \\ (h', \mathsf{v}, (\mathsf{v}_l, cbody(C);\ \texttt{return this}) \circ (\mu, \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}^b) \end{array}}$$

$$[\textsc{BlkBeg}]\ \frac{\mathsf{v}_l = \{\overline{x} \mapsto ival(\overline{T})\}}{(h, \mathsf{v}, (\mu, \{\overline{T}\ \overline{x}; stmt\};\ mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}, (\mathsf{v}_l \cdot \mu, stmt;\ \texttt{BE}\ mc) \circ \mathsf{CS}^b)}$$

$$[\textsc{BlkEnd}]\ (h, \mathsf{v}, (\mathsf{v}_l \cdot \mu, \texttt{BE}\ mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}, (\mu, mc) \circ \mathsf{CS}^b)$$

$$[\textsc{Whl}_1]\ \frac{[\![e]\!]_h^{\mathsf{v},\mu}}{(h, \mathsf{v}, (\mu, \texttt{while}\ (e)\ \{stmt\}; mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}, (\mu, stmt;\ \texttt{while}\ (e)\ \{stmt\}; mc) \circ \mathsf{CS}^b)}$$

$$[\textsc{Whl}_2]\ \frac{\neg [\![e]\!]_h^{\mathsf{v},\mu}}{(h, \mathsf{v}, (\mu, \texttt{while}\ (e)\ \{stmt\}; mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}, (\mu, mc) \circ \mathsf{CS}^b)}$$

$$[\textsc{Cond}_1]\ \frac{[\![e]\!]_h^{\mathsf{v},\mu}}{(h, \mathsf{v}, (\mu, \texttt{if}\ (e)\ \{stmt_1\}\ \texttt{else}\ \{stmt_2\};\ mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}, (\mu, stmt_1;\ mc) \circ \mathsf{CS}^b)}$$

$$[\textsc{Cond}_2]\ \frac{\neg [\![e]\!]_h^{\mathsf{v},\mu}}{(h, \mathsf{v}, (\mu, \texttt{if}\ (e)\ \{stmt_1\}\ \texttt{else}\ \{stmt_2\};\ mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}, (\mu, stmt_2;\ mc) \circ \mathsf{CS}^b)}$$

$$[\textsc{Ret}]\ \frac{(\mathsf{v}', \mu_2') = vupd(\mathsf{v}, \mu_2, x \mapsto [\![e]\!]_h^{\mathsf{v},\mu_2})}{(h, \mathsf{v}, (\mu_1, \texttt{return}\ e) \circ (\mu_2, \texttt{rcv}\ x;\ mc) \circ \mathsf{CS}^b) \rightsquigarrow (h, \mathsf{v}', (\mu_2', mc) \circ \mathsf{CS}^b)}$$

Table 2.7: Simple *Java*-like language: operational semantics

The rules BlkBeg and BlkEnd deal with the introduction and removal of block variable functions due to a block statement. The rule BlkBeg does not only add a new variable function to the variable function list of the top most activation record but in the code of the record it also puts an auxiliary symbol (`BE`) at the end of the block statement, in order to mark the end of the block's scope. Then the counterpart of BlkBeg, namely BlkEnd, removes `BE` and its associated variables function when it is the topmost statement.

The while-loop is processed by either removing the while-loop from the active activation record or by extending it with a copy of the while-loop's body statement – depending on the evaluation of the while-loop's condition expression. In a similarly straightforward manner, the conditional statement is reduced to one of its sub-statements.

The RET rule is applied when the topmost activation record consists of a return term, only. The record as well as the receive statement of the calling activation record is removed such that the calling record becomes the topmost active record. Moreover, the caller's local variable list and the global variable list is updated by the return value.

Remark 2.3.2: Some rules of the operational semantics depend on the program code. Rule CALL, for instance, extends the call stack by the method body of the callee class. Thus, the transition rules are to be understood in context of a given program p and consequently the transition arrow should be annotated by the program: $\rightsquigarrow_p$. In most cases, however, we omit the annotation.

Definition 2.3.3 (Program execution)**:** Let

$$p \equiv \overline{T}\,\overline{x};\ \overline{cldef}\ \{stmt;\ \texttt{return}\}$$

be a syntactically correct and well-typed program of our language. A *program execution* of p is a finite sequence of reduction steps starting from the *initial configuration* of the program

$$c_{init}(p) \stackrel{\text{def}}{=} (h_\perp, \{\overline{x} \mapsto ival(\overline{T})\}, (\mathsf{v}_\perp, stmt;\ \texttt{return})),$$

where $h_\perp$ denotes the empty heap and $\mathsf{v}_\perp$ the empty local variable function. That is, both functions are completely undefined.

If we are interested neither in the exact length of a finite reduction sequence nor in its intermediate configurations we use a transition arrow annotated with the Kleene star,

$$c_{init}(p) \rightsquigarrow^* c,$$

expressing that there exists a finite sequence of reduction steps from the initial configuration to the configuration c or that both configurations are identical. In other words, we use the Kleene star annotation to refer to the reflexive and transitive closure of the semantics transition relation. If the call stack of c consists only of the last return statement of the main body, then we call c a *terminal configuration*. If otherwise c cannot be reduced any further, we call the configuration *faulty*.

2.4 Extension by components: the *Japl* language

As mentioned in the introduction, the basic idea for unit or component testing is to test the component in isolation. The production code that represents the *environment* of the component is replaced by some test code which investigates the component by interacting with it. Since we aim at a model-driven component testing approach where component tests are usually derived from formal, hence

rather abstract, specifications, we are in particular interested in testing techniques where a test does not rely on or aim at implementation details of the component but where a test only deals with the component's effects on its environment. In order to investigate the means by which a component might have an effect on its environment, we extend our *Java*-like language with constructs that allow for discriminating component and environment code. This is done by integrating a notion of *components* into our language. A component is basically a set of classes. Classes of one component can be imported by another component (or by the program). A crucial point is here that importing a class is not realized by importing the *code* of the class definition. Instead, we formalize the operational semantics of a program in absence of the code of imported classes. This enables us to identify the most general characterization of a component's influence on its environment, only assuming the interfaces of its classes.

In this section we will extend the *Java*-like language with the notion of components. The extended language will be used in subsequent chapters where we will refer to it as the *Japl programming language*. Conceptually, *Japl* supports components, in that it allows programs to import externally defined classes. This means, a program might instantiate and call methods of classes which are not part of the program's code but are assumed to exist in some other component. While the entailed syntax and typing modifications are quite simple, the operational semantics has to be given in form of an *open semantics*. In other words, we have to formulate the operational semantics without the code of the externally defined classes. The section is followed by a formal description of our testing approach given in context of our extended language.

2.4.1 Syntax

The extension of the syntax is very simple and straightforward, as can be seen in Table 2.8. We merely add a construct for declaring imported classes which introduces the name of the class only. For the sake of simplicity we do not introduce name spaces, but instead we assume that the names of all imported and all locally defined classes are different. The grammar introduces a new non-terminal symbol p' which replaces p of the former grammar. An element of p' is called a *component*. Thus, the program does not only import classes of other components but it constitutes a component itself. In particular, all components contain a main body. This is again due to simplicity, because otherwise we would have to differentiate components and programs (and in this case only component classes could be imported but a component could not import a program class). However, we will see in the operational semantics that only the main body of one single component is in execution.[2] In the following we will use the word *program* in order to refer to the component whose code is given and processed in the operational semantics. Note

[2]Assuming that each component provides its own main body is comparable to the widely used technique to equip a *Java* package with a static main method allowing for the stand-alone execution of the package due to testing or demonstration purposes.

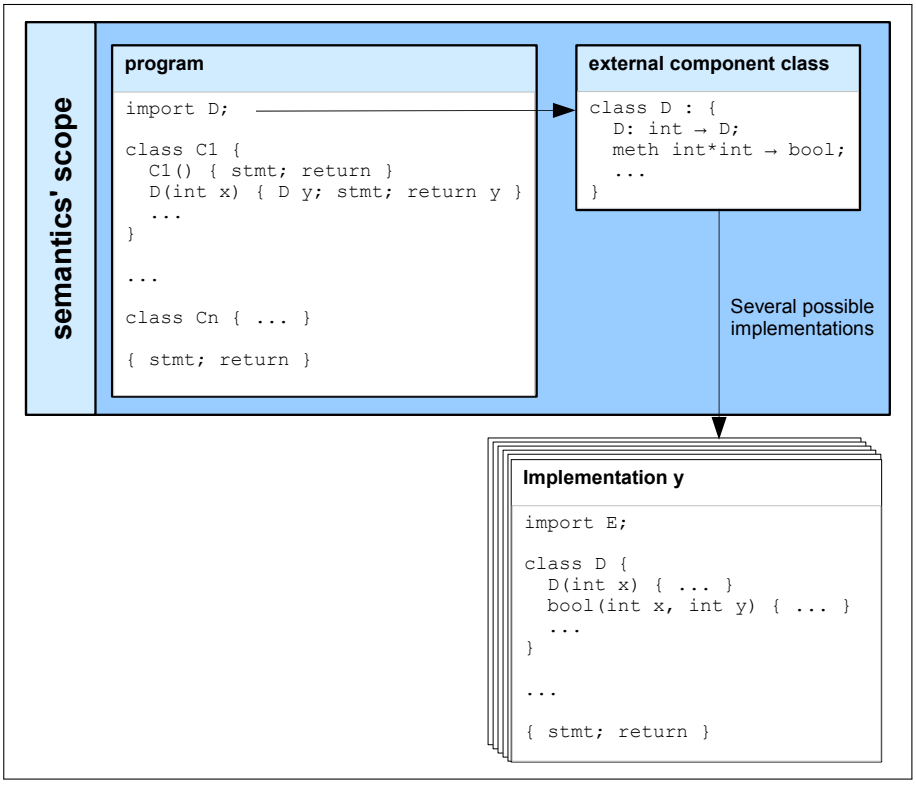

Figure 2.1: Notion of component

that this doesn't necessarily mean that the main body of the program is executed, but maybe only the code of its classes is subject of the operational semantics. The program (component) might use classes of some *external* components. However, if the context is clear or if we don't want to be specific we sometimes speak only of components. The notion of components is depicted in Figure 2.1. It shows a program which defines classes C_1 to C_n as well as a main body and it additionally imports another class D from an unspecified external component. Thus, executing the given program, the operational semantics does not know the implementation but only the type of D.

p'	$::= \overline{impdecl};\ p$	program/component
$impdecl$	$::= \texttt{import}\ C$	import

Table 2.8: *Japl* language : syntax

$$[\text{T-PROG'}]\frac{\Gamma' = \Gamma, \overline{x}{:}\overline{T} \quad \Theta = \mathit{cltype}(\overline{\mathit{cldef}}) \qquad \Gamma;\Delta \vdash \overline{\mathit{impdecl}} : \mathtt{ok} \quad \Gamma';\Delta,\Theta \vdash \overline{\mathit{cldef}} : \mathtt{ok} \quad \Gamma';\Delta,\Theta \vdash \mathit{stmt} : \mathtt{ok}}{\Gamma;\Delta \vdash \overline{\mathit{impdecl}};\ \overline{T}\,\overline{x};\ \overline{\mathit{cldef}}\ \{\mathit{stmt};\ \mathtt{return}\} : \Theta}$$

$$[\text{T-IMPORT}]\frac{C \in \mathit{dom}(\Delta)}{\Gamma;\Delta \vdash \mathtt{import}\ C : \mathtt{ok}}$$

Table 2.9: *Japl* language: type system (stmts)

2.4.2 Static Semantics

We want to allow "cross-importing", i.e., components should be able to mutually import their classes. To this end, we have to reformulate the typing judgment on the program/component level such that it does not just state the program's well-typedness but it also explicitly mentions the program's classes committed to its environment in terms of a type mapping Θ. Moreover, we require that the assumed type context Δ of a program's type check already includes the types of the imported (*assumed*) classes. In other words, a program is now type-checked in an assumption-commitment context as it can be seen in typing rule T-PROG' in Table 2.9. This is closely related to the required and provided interfaces in *UML* compoment diagrams[65].

Finally, we have to add a new rule for the import construct. However, since the import construct only mentions the name of the class but no further typing information, we only have to check whether the imported class name is in the domain of Δ. All other rules of Table 2.2 and Table 2.3 remain the same.

As open programs are now typed in an assumption-commitment context, we have to reformulate the well-typedness definition for program.

Definition 2.4.1 (Well-typedness)**:** An open program p' is well-typed if there exist type mappings Θ and Δ such that the judgment

$$;\Delta \vdash p' : \Theta$$

is derivable. Similar to Definition 2.2.2, we demand an empty local type context for p. Therefore, well-typedness of a program p regarding the assumption-commitment context Δ, Θ is denoted by

$$\Delta \vdash p' : \Theta\ .$$

Remark 2.4.2 (Accessibility of global variables)**:** The global variables of a component are not "published" in the component's commitment context. The important consequence is that global variables of a component are not accessible by other components but they are always global with respect to the defining component, only.

2.4.3 Operational Semantics

The introduction of the import construct leads to an under-specification of the program: only the names and the types of the imported classes are given but not the code. The consequence is that the semantics definition of a component now consists of two parts. The first part, called *internal semantics*, deals with *internal* computations only, i.e., computations which are completely independent of the imported classes but solely determined by the program's code. This part is given in form of a transition semantics which is almost identical to the one already given in Table 2.7. We only add a premise in rule CALL and in rule NEW which ensures that the called method or constructor, respectively, indeed belongs to a class of the given program code. As for rule NEW, this additional check is very simple, since the class name itself is part of the `new` statement. As for the method call, we have to find out, if the callee object, named o, is an instance of a program class. We will see later, however, that the heap function stores information about objects of program classes, only. Therefore, the check can be easily realized by adding the premise $o \in dom(h)$.[3]

Labeled transition system. The second part of the operational semantics is called *external semantics*. It deals with computation steps that involve an (instance of an) external class. These steps must be handled differently as we do not have the code of the external classes: if, for example, the program calls a method of an external class, then we cannot use rule CALL, because lacking the method's code we cannot copy its body on the call stack in order to execute it afterwards. Instead, due to synchronous message passing, no further internal computations are possible right after the corresponding transition has been taken, as the transition gives away the control to an external component. We call this kind of transitions *outgoing* communication or computation steps. Right after an outgoing computation step, no transition can be taken unless it involves the return of the control back to the program. Generally speaking, we call transitions that entail the transition of control from an external component to the program *incoming* communication or *incoming* computation steps. In our method call example such an incoming communication could be either an immediate return from the called method of the external component or a call of a method of the program. Intuitively speaking, outgoing communication is due to a program statement, incoming communication is caused by an external component.

The external semantics is formalized by a *labeled transition system*, that is, a transition system where each transition is annotated with a label. Each transition of the external semantics represents an *interface communication*, i.e., a communication between the program and an external component and the transition's label holds the details about this communication. We have already seen that the

[3] In fact, it is not even necessary to add these checks, as $h(o).class$ in rule CALL and $cparams(C)$ in rule NEW are anyway undefined for an external object o, and, resp., class C. Adding the premises though makes the requirement more explicit.

different kinds of interface communications can be divided up into two groups. On the one hand, outgoing communications are provoked by the program giving the control to the external component. There exist three constructs in the programming language that may cause a hand-over of the control:

- a method call of an instance of an imported class,
- a constructor call of an imported class, and
- a return from a program method that was previously called by an external component.

To indicate the outgoing character, the transition labels of these communications are decorated with an exclamation mark (!). On the other hand, there are three possible kinds of communication that pass the control from an external component on to the program:

- a method call of an instance of a program class,
- a constructor call of a program class, and
- a return from a method of an imported class which was previously called by the program.

The labels of these incoming communication steps are decorated with a question mark (?). Summarizing, the possible interface interactions are method calls and returns as well as constructor call and returns which are either outgoing or incoming. Additionally, for each interface communication we explicitly have to point out the involved object names which pass the interface for the first time during the program execution. To this end, each communication label that propagates new names is equipped with a ν-binder indicating the new names introduced by the label. In particular, the ν-binder provides these names in terms of a type mapping Θ, since we are also interested in their types. The reason for this will be given later. Thus, the set of communication, or transition, labels a is given by the following grammar:

$$
\begin{aligned}
&a ::= \gamma? \mid \gamma! \\
&\gamma ::= \langle \mathit{call}\ o.m(v,\ldots,v)\rangle \mid \langle \mathit{new}\ C(v,\ldots,v)\rangle \mid \langle \mathit{return}\ (v)\rangle \mid \nu(\Theta).\gamma, \\
&\quad \text{where } o \in N, v \in \mathit{Vals}, \text{ and } \Theta \text{ is a type mapping.}
\end{aligned}
$$

Restrictions due to realizability. The outgoing method call example above has shown, that the exact reaction of the external component is not determined; in general it is not known whether the component immediately returns a value or calls back a program method. Apart from the types, the program also has no control on the values that are involved in an incoming communication. In

fact, this is why we need a labeled transition system to formalize the external semantics: the details about an incoming communication cannot be deduced from the program code but are introduced by the communication label. More generally speaking, regarding incoming communications, the *operational semantics* is non-deterministic – although the *programming language* Japl in itself is deterministic. Despite the absence of the code of the external components, however we want to restrict the non-determinism of the operational semantics such that we exclude incoming communication which could not be carried out by any component that is written in our language *Japl*. This entails a number of requirements which have an influence on the structure of the transition rules:

well-typedness: Since *Japl* is strongly typed, a valid program written in *Japl* could never implement a wrongly typed method or constructor call. This fact implies a dual requirement for communication imposed by an external component. Specifically, the semantics shall only permit incoming calls of methods and constructors that are indeed provided by the program code. This requirement is often phrased as "no message-not-understood error". Also the number and the types of the incoming call's actual parameters must comply with the method or, respectively, constructor definition.

Likewise, the typing rules of our language ensure that the return value provided by a method body is always of the return type stipulated in the method's signature. Again, the dual requirement is that the semantics has to exclude wrongly-typed incoming return values.

consistent information flow: Components do not share variables but values can only be communicated between two components via method or constructor calls (and its corresponding returns). In particular, a component can only use an object if either it is an instance of one of the component's own classes or if the class-providing component has previously passed the corresponding object name in context of a method or constructor call between the two components. Thus, the external semantics has to ensure that an incoming communication may mention an object name of a program class only if previously the program has passed the name to the component by means of an outgoing communication.

consistent control flow: We have explained that an outgoing communication disables the reduction of the program unless an incoming communication occurs. Dually, the external semantics may only allow an incoming communication if the external component indeed has the control at that moment.

balance: The syntax definition of our language allows a return term only to appear exactly once at the very end of method and constructor bodies. That is, the execution of such return term is always preceded by a call of the corresponding method or constructor. For the external semantics, this means that an incoming return may only happen, if previously a matching outgoing call was invoked.

To meet the first two requirements, transitions of the external semantics are formalized in an assumption-commitment context, similar to the program's typing judgments of the static semantics. In the external semantics, however, the contexts also have a dynamic aspect, in the sense that the commitment context Θ does not only include the committed class names but also the names of their instances that have been passed on to the component during the execution of the program. Dually, the assumption context Δ consists of the component's class and object names passed on to the program. Thus, transitions representing computation steps of the external semantics will follow the following scheme:

$$\Delta \vdash c : \Theta \xrightarrow{a} \Delta' \vdash c' : \Theta',$$

where a is the label describing the communication and c is the configuration of the program prior to the transition. We used primed versions of Δ, Θ , and c in the transition scheme to indicate that the configuration as well as the context might change due to the transition. In particular, since the name contexts, Θ and Δ, keep track of the object names that passed the interface, each transition of the external semantics causes an extension of the context by exactly the names that are mentioned in the ν-binder of the transition's communication label. The transition rules of the external semantics are again formalized in terms of a deduction system where the conclusion of each rule consists of a transition scheme. Moreover, most of the rules are equipped with some premises defining the rule's application condition. In particular, we have to add certain premises to meet the realizability requirements regarding incoming communication.

To ensure a consistent information flow, the transition rules for incoming communication require that each object name mentioned in the call or return, respectively, is either included in the commitment context, in the assumption context or in the name context of the communication label's ν-binder. The first two cases indicate that the program and the component have communicated the object earlier already; the third case represents the situation where the object name shows up at the interface for the first time. Since all names in the contexts are also accompanied by their types, the contexts can be used the check for well-typedness as well. To keep the definition of the external semantics concise we encapsulate the type and information flow check in an auxiliary notation of the following form:

$$\Delta \vdash a : \Theta,$$

which expresses that, within a context represented by Δ and Θ, the computation step represented by a conforms to well-typedness and a consistent information flow. The definition is given in Table 2.10. The rule T-CallI deals with the type and information flow check of a label that represent an incoming method call. This is basically carried out by the premises of the first row: the first premise states that the callee object is indeed an object of the program; the second and the third premise ensure the existence of the method within the callee class as well as the well-typedness of the actual parameters. Since we use the name contexts for the type check we ensure at the same time that all object names of the communication

$$[\text{T-CallI}]\ \frac{\begin{array}{c} C = \Theta(o) \quad \overline{T} = \Theta(C)(m).dom \quad \Theta, \Delta, \Delta' \vdash \overline{v}{:}\overline{T} \\ dom(\Delta') \perp dom(\Delta, \Theta) \qquad dom(\Delta') \subseteq \overline{v} \end{array}}{\Delta \vdash \nu(\Delta').\langle call\ o.m(\overline{v})\rangle ? : \Theta}$$

$$[\text{T-NewI}]\ \frac{\begin{array}{c} \Theta(C)(C).dom = \overline{T} \quad \Theta, \Delta, \Delta' \vdash \overline{v}{:}\overline{T} \\ dom(\Delta') \perp dom(\Delta, \Theta) \quad dom(\Delta') \subseteq \overline{v} \end{array}}{\Delta \vdash \nu(\Delta').\langle new\ C(\overline{v})\rangle ? : \Theta}$$

$$[\text{T-RetI}]\ \frac{dom(\Delta') \perp dom(\Delta, \Theta) \quad dom(\Delta') \subseteq v}{\Delta \vdash \nu(\Delta').\langle return(v)\rangle ? : \Theta}$$

Table 2.10: Label check for incoming communication

label are mentioned in one of the name contexts, hence, a consistent information flow is assured. The premises of the second row check for well-formedness of the ν-bound name context Δ': first, the names which are claimed to pass the interface for the first time, must not be included in Δ or Θ. We use $\perp$ to express that two sets are disjunct. Second, the name context Δ' must not include more names than actually communicated by this interface communication.

The rule T-NewI deals with an incoming constructor call and follows the scheme of rule T-CallI. However, we need not to look up the name of the invoked class. Finally, the check of an incoming return label even only consists of the well-formdness check of the ν-bound name context.

Configurations. We have learned that extending the language with the notion of components leads to a new kind of method (and constructor) calls: besides internal calls, where caller and callee belong to the same component, we now also have external (or cross-border) calls, where caller and callee sit in different components. With respect to the configurations of a program, this means, we also have to introduce a second receive statement enabling us to distinguish activation records that are blocked due to an internal call from activation records that are blocked due to an external call. More precisely, regarding internal calls we keep using the receive statement that we have introduced previously. As for the external calls we use a similar receive statement but which is additionally annotated with the return type of the expected return value. Thus, we now have three types of activation records:

$$\mathsf{AR}^a ::= (\mu, mc) \quad \mathsf{AR}^{ib} ::= (\mu, \texttt{rcv}\ x;\ mc) \quad \mathsf{AR}^{eb} ::= (\mu, \texttt{rcv}\ x{:}T;\ mc).$$

For the sake of convenience we will also use AR for activation records in general and AR^b for internally or externally blocked activation record. Moreover, we will use AR^i for AR^a and AR^{ib} records, i.e., for activation records whose next upcoming reduction will be due to an internal step.

We redefine the call stack structure of configurations of components:

$$\mathsf{CS}^a ::= \mathsf{AR}^a \circ \mathsf{CS}^b \qquad \mathsf{CS}^b ::= \mathsf{CS}^{ib} \mid \mathsf{CS}^{eb}$$
$$\mathsf{CS}^{eb} ::= \mathsf{AR}^{eb} \circ \mathsf{CS}^b \mid \epsilon \qquad \mathsf{CS}^{ib} ::= \mathsf{AR}^{ib} \circ \mathsf{CS}^b$$
$$\mathsf{CS} ::= \mathsf{CS}^a \mid \mathsf{CS}^{eb}$$

Contrary to the call stack of a closed program, a call stack of an open program does not necessarily have an *active* activation record on top but the call stack can also be externally blocked due to an outgoing call. Note that externally blocked call stacks also include the *empty* call stack. The reason is that in some cases not the main body of the program but the main body of an external component is executed. Then, for instance at the very beginning of the execution, we start with a configuration that entails an empty call stack.

Furthermore, we introduce the notion of well-typedness of configurations. Similar to the typing judgments of open programs, well-typedness of a configuration is expressed by writing the configuration in an assumption-commitment context consisting of type mappings Δ and Θ. Before we present the definition for well-typed configurations, we define the free variables of activation record code.

Definition 2.4.3 (Free variables)**:** Assume code mc of an activation record. The expression $fvars(mc)$ denotes the set of *free variables* within mc. The function $fvars$ is given for activation record code by the recursive definition shown in Table 2.11.

Definition 2.4.4 (Well-typed configuration)**:** Let $(h, \mathsf{v}, \mathsf{CS}) \in \mathit{Conf}$ be a configuration and Δ, Θ typing contexts. We say the configuration $(h, \mathsf{v}, \mathsf{CS})$ is well-typed in context of the assumed type mapping Δ and the committed type mapping Θ if the following properties hold:

1. For all $(C, f) \in ran(h)$, it is $\Theta \vdash C[\![\ldots]\!]$.
2. For all $(\mu, mc) \in \mathsf{CS}$ and for all $x \in fvars(mc)$, it is $x \in dom(\mathsf{v}) \cup dom(\mu)$.
3. For all $\mu_m \in \mathsf{CS}$, it is $[\![\texttt{this}]\!]_{\_}^{\mu_m,\_} \in dom(h)$.

Thus, well-typedness of configurations ensures that the class names of objects in the heap are indeed committed as names of classes. The second statement ensures that each variable mentioned in the code of an activation record is either a global variable or a local variable of the record. We use $dom(\mathsf{v}_1 \cdot \ldots \cdot \mathsf{v}_k)$ as abbreviation for $dom(\mathsf{v}_1) \cup \ldots \cup dom(\mathsf{v}_k)$. Finally, we are assured that the local variables of each activation record representing a method instance provide a value for the self-reference `this` which in turn refers to an object in the heap.

$$
\begin{array}{rcl}
\mathit{fvars}(\texttt{rcv}\ x; mc) & \stackrel{\text{def}}{=} & \{x\} \cup \mathit{fvars}(mc) \\
\mathit{fvars}(\texttt{rcv}\ x{:}T; mc) & \stackrel{\text{def}}{=} & \{x\} \cup \mathit{fvars}(mc) \\
\mathit{fvars}(x = e) & \stackrel{\text{def}}{=} & \{x\} \cup \mathit{fvars}(e) \\
\mathit{fvars}(f = e) & \stackrel{\text{def}}{=} & \mathit{fvars}(e) \\
\mathit{fvars}(x = e.m(e_1, \ldots, e_k)) & \stackrel{\text{def}}{=} & \{x\} \cup \mathit{fvars}(e) \cup_{i=1,\ldots,k} \mathit{fvars}(e_i) \\
\mathit{fvars}(x = \texttt{new}\ C(e_1, \ldots, e_k)) & \stackrel{\text{def}}{=} & \{x\} \cup_{i=1,\ldots,k} \mathit{fvars}(e_i) \\
\mathit{fvars}(\mathit{stmt}_1;\ \mathit{stmt}_2) & \stackrel{\text{def}}{=} & \mathit{fvars}(\mathit{stmt}_1) \cup \mathit{fvars}(\mathit{stmt}_2) \\
\mathit{fvars}(\{\overline{T}\,\overline{x};\ \mathit{stmt}\}) & \stackrel{\text{def}}{=} & \mathit{fvars}(\mathit{stmt}) \setminus \overline{x} \\
\mathit{fvars}(\texttt{while}(e)\{\mathit{stmt}\}) & \stackrel{\text{def}}{=} & \mathit{fvars}(e) \cup \mathit{fvars}(\mathit{stmt}) \\
\mathit{fvars}(\texttt{if}(e)\{\mathit{stmt}_1\}\ \texttt{else}\ \{\mathit{stmt}_2\}) & \stackrel{\text{def}}{=} & \mathit{fvars}(e) \cup \mathit{fvars}(\mathit{stmt}_1) \cup \mathit{fvars}(\mathit{stmt}_2) \\
\mathit{fvars}(\texttt{BE}) & = & \emptyset \\
\mathit{fvars}(\texttt{return}\ e) & = & \mathit{fvars}(e) \\
\mathit{fvars}(\texttt{return}) & = & \emptyset \\
\mathit{fvars}(x) & \stackrel{\text{def}}{=} & \{x\} \\
\mathit{fvars}(\texttt{this}) & \stackrel{\text{def}}{=} & \emptyset \\
\mathit{fvars}(f) & \stackrel{\text{def}}{=} & \emptyset \\
\mathit{fvars}(\texttt{null}) & \stackrel{\text{def}}{=} & \emptyset \\
\mathit{fvars}(\texttt{op}(e_1, \ldots, e_k)) & \stackrel{\text{def}}{=} & \mathit{fvars}(e_1) \cup \ldots \cup \mathit{fvars}(e_k)
\end{array}
$$

Table 2.11: Free variables

Transition rules. After this somewhat longer preliminary explanation, let us now have a closer look at the rules of the external semantics. They are given in Table 2.12. To improve readability, we distinguish conditions that express a real limitation of the rule's application from conditions that only introduce variables used to keep the definition short. The first kind of conditions are written as premises, the latter are written as side conditions. An exception is the introduction of the communication label a which is always listed as the first premise in every rule.

The first rule, CallO, implements an outgoing call. This rule must be applied only if the callee of the method is indeed an object of an external component. This is checked by assuring that the callee object name is an element of the domain of the name context Δ. In this sense, rule CallO is the counterpart of rule Call which deals with internal method calls, only. However, we do not put a method body on the call stack but instead we add a `rcv` statement to the current activation

$$[\textsc{CallO}]\ \frac{a = \nu(\Theta').\langle call\ o.m(\overline{v})\rangle!\quad o \in dom(\Delta)}{\begin{array}{l}\Delta \vdash (h, \mathsf{v}, (\mu, x = e.m(\overline{e}); mc) \circ \mathsf{CS}^b) : \Theta \xrightarrow{a} \\ \Delta \vdash (h, \mathsf{v}, (\mu, \texttt{rcv}\ x{:}T;\ mc) \circ \mathsf{CS}^b) : \Theta, \Theta'\end{array}}$$

where $o = [\![e]\!]_h^{\mathsf{v},\mu}$, $\overline{v} = [\![\overline{e}]\!]_h^{\mathsf{v},\mu}$, $T = \Delta^2(o)(m).ran$, and $\Theta' = \mathsf{new}(h, \overline{v}, \Theta)$

$$[\textsc{NewO}]\ \frac{a = \nu(\Theta').\langle new\ C(\overline{v})\rangle!\quad C \in dom(\Delta)}{\begin{array}{l}\Delta \vdash (h, \mathsf{v}, (\mu, x = \texttt{new}\ C(\overline{e}); mc) \circ \mathsf{CS}^b) : \Theta \xrightarrow{a} \\ \Delta \vdash (h, \mathsf{v}, (\mu, \texttt{rcv}\ x{:}C; mc) \circ \mathsf{CS}^b) : \Theta, \Theta'\end{array}}$$

where $\overline{v} = [\![\overline{e}]\!]_h^{\mathsf{v},\mu}$ and $\Theta' = \mathsf{new}(h, \overline{v}, \Theta)$

$$[\textsc{RetO}]\ \frac{a = \nu(\Theta').\langle return(v)\rangle!}{\begin{array}{l}\Delta \vdash (h, \mathsf{v}, (\mu, \texttt{return}\ e) \circ \mathsf{CS}^{eb}) : \Theta \xrightarrow{a} \\ \Delta \vdash (h, \mathsf{v}, \mathsf{CS}^{eb}) : \Theta, \Theta'\end{array}}$$

where $v = [\![e]\!]_h^{\mathsf{v},\mu}$ and $\Theta' = \mathsf{new}(h, v, \Theta)$

$$[\textsc{CallI}]\ \frac{a = \nu(\Delta').\langle call\ o.m(\overline{v})\rangle?\quad \Delta \vdash a : \Theta}{\begin{array}{l}\Delta \vdash (h, \mathsf{v}, \mathsf{CS}^{eb}) : \Theta \xrightarrow{a} \\ \Delta, \Delta' \vdash (h, \mathsf{v}, (\mathsf{v}_l, mbody(C, m)) \circ \mathsf{CS}^{eb}) : \Theta\end{array}}$$

where $C = \Theta(o)$, $\overline{T}\ \overline{x} = mparams(C, m)$, $\overline{T'}\ \overline{x'} = mvars(C, m)$, and $\mathsf{v}_l = \{\texttt{this} \mapsto o, \overline{x} \mapsto \overline{v}, \overline{x'} \mapsto ival(\overline{T'})\}$

$$[\textsc{NewI}]\ \frac{a = \nu(\Delta').\langle new\ C(\overline{v})\rangle?\quad \Delta \vdash a : \Theta}{\begin{array}{l}\Delta \vdash (h, \mathsf{v}, \mathsf{CS}^{eb}) : \Theta \xrightarrow{a} \\ \Delta, \Delta' \vdash (h', \mathsf{v}, (\mathsf{v}_l, cbody(C)) \circ \mathsf{CS}^{eb}) : \Theta\end{array}}$$

where $o \in N \setminus dom(h)$, $h' = h[o \mapsto Obj_\perp^C]$, $\overline{T}\ \overline{x} = cparams(C)$, $\overline{T'}\ \overline{x'} = cvars(C)$, and $\mathsf{v}_l = \{\texttt{this} \mapsto o, \overline{x} \mapsto \overline{v}, \overline{x'} \mapsto ival(\overline{T'})\}$

$$[\textsc{RetI}]\ \frac{a = \nu(\Delta').\langle return(v)\rangle?\quad \Delta \vdash a : \Theta\quad \Delta, \Delta', \Theta \vdash v : T}{\begin{array}{l}\Delta \vdash (h, \mathsf{v}, (\texttt{rcv}\ x{:}T;\ mc) \circ \mathsf{CS}^b) : \Theta \xrightarrow{a} \\ \Delta, \Delta' \vdash (h, \mathsf{v}', (\mu', mc) \circ \mathsf{CS}^b) : \Theta\end{array}}$$

where $(\mathsf{v}', \mu') = vupd(\mathsf{v}, \mu, x \mapsto v)$

Table 2.12: *Japl* language: operational semantics (ext.)

record which is annotated with the return type of the called method. Thus, the activation record becomes an externally blocked activation record. To determine the return type, we consult the name context Δ. Since, however, the object's type is a class name, we have to apply Δ twice to get the corresponding class type which in turn yields the return type of the method. We use Δ^2 to express the double application of Δ. The return type will be needed in rule RetI to check that the incoming return value matches the outgoing call.

Apart from modifying the call stack we also have to update the *committed* name context, since the outgoing call might involve the propagation of names of some program objects which haven't passed the interface previously. In order to find out these new object names we use an auxiliary function $\mathsf{new}(h, \overline{o}, \Theta)$. For a given set of objects $\overline{o}$ it consults the heap h in order to create a subset of $\overline{o}$ which consists only of instances of program classes accompanied by their types. Finally the current committed name context Θ is subtracted leaving only the new object names. Its definition is as follows:

Definition 2.4.5 (New names propagation)**:**

$$\mathsf{new}(h, \overline{o}, \Theta) \stackrel{\text{def}}{=} \{(o{:}C) \mid o \in \overline{o} \wedge o \in dom(h) \wedge C = h(o).class\} \setminus \Theta.$$

The rule NewO deals with the instantiation of a class of an external component. It resembles rule CallO but it is a bit simpler, as there is no need to look up the return type. Rule RetO processes a return statement. More specifically, it must only deal with outgoing returns, i.e., it must only be applied if the corresponding method has been called by an external component. This, however, is the case, if the next activation record on the call stack is an externally blocked record but, in particular, not an internally blocked record that waits for an internal return. Thus, the rule expects that the topmost activation record is followed by an externally blocked call stack.

The next three rules deal with incoming communications, namely with incoming method calls (CallI), incoming object creation (NewI), and incoming returns (RetI). All three rules implement the type and information flow consistency check as it has been previously explained. Moreover, all three rules assert consistency of the control flow, in that they start from a configuration with an externally blocked call stack. In Rule RetI, however, we do not use CS^{eb} in the judgment to express this requirement but we explicitly demand the annotated `rcv` statements on top of the call stack, as we use its annotated return type to check well-typedness of the return value in the communication label. Second, an annotated `rcv` statement on top also ensures the balance condition, as it indicates a preceded outgoing call (cf. rules CallO and NewO). After all, CS^{eb} also comprises the empty statement, hence, in contrast to Rule RetI, the Rules CallI and NewI can also be applied when the call stack is empty. This allows to execute the program through the use of an external main body.

Note that our approach for defining the external semantics of *Japl* is closely related to Tretmans' input-output transition systems[69]. This kind of labeled transition systems also distinguishes input and output labels in order to determine the direction of communication occurring between a component and its environment. Moreover, input-output transition systems are input-enabled which reflects the fact that an exact behavior of the environment is not specified. However, although *Japl* components are generally input-enabled as well we have made clear that realizability demands some restrictions regarding the possible interface communication. For instance, as already mentioned the balance condition forbids transitions with incoming return labels in certain situations.

Remark 2.4.6 (Renaming)**:** The introduction of new objects of an external class by means of an incoming communication is actually carried out by passing the object's *name* to the program. Although the label check for incoming communication ensures that the name of such an object indeed hasn't shown up *at the interface* previously, there might exist an *internal* object of the same name within the program already. In this case, we assume an implicit ad hoc renaming of the internal object. More precisely, we rename the object in the heap and at the same time change all corresponding references within

fields or variables of the component. As an important consequence in the following we will consider configuration equality always up to renaming of objects, only.

Note, that we do not have to arrange for capture-free renaming as object names are globally[4] unique.

An important feature of the operational semantics is that it preserves the well-typedness of configurations. This is formalized in the following lemma.

Lemma 2.4.7 (Subject reduction)**:** Assume $\Delta_0 \vdash p' : \Theta_0$ and $\Delta \vdash c : \Theta$ for a configuration c such that Δ and Θ represent extensions of Δ_0 and Θ_0, respectively.

1. If $c \rightsquigarrow_p c'$ then also $\Delta \vdash c' : \Theta$.

2. If $\Delta \vdash c : \Theta \xrightarrow{a}_p \Delta' \vdash c' : \Theta'$ then $\Delta' \vdash c' : \Theta'$. Moreover, Δ' and Θ' again represent extensions of Δ_0 and Θ_O.

Thus, the subject reduction lemma justifies the use of typing judgments for transition states of the external operational semantics.

Definition 2.4.8 (Program execution; program traces)**:** Let

$$p' \equiv \overline{impdecl}\ \overline{T}\ \overline{x};\ \overline{cldef}\ \{stmt;\ \texttt{return}\}$$

be an open program with $\Delta \vdash p' : \Theta$. We broaden the application of c_{init}, defined in Definition 2.3.3, such that we also apply it to open programs p'. However, in the context of open programs we call $c_{init}(p')$ an *active* initial configuration. Dually, we introduce a *passive* initial configuration

$$\overline{c_{init}}(p') \stackrel{\text{def}}{=} (h_\perp, \overline{x} \mapsto ival(\overline{T}), \epsilon),$$

where the call stack is not initialized with the main body of p' but it is empty, meaning that the main body of an external component is to be executed.

The execution of an open program is represented by a finite, possibly empty, sequence of internal and external transitions starting from one of its initial configurations. The sequence of communication labels arising from a program execution is called an *(observable interaction) trace* of the program. We use an annotated arrow $\stackrel{t}{\Longrightarrow}$ to represent a program execution that implements the trace t. The corresponding rules are given in Table 2.13. Thus, the execution of an open program represents the reflexive transitive closure of the internal and external transitions.

From now on, we deal with open programs only. Specifically, a closed program is considered to be an open program which does not import external classes. Therefore, the syntactical discrimination between open and closed programs by using different non-terminal symbols p and p' is not necessary anymore, instead we will use p also for open programs.

[4]With "global" we always mean global with respect to a certain component only, but not to the whole system consisting of all components that might be involved in a program execution.

$$[\textsc{Intern}]\ \frac{c \longrightarrow^* c'}{\Delta \vdash c : \Theta \stackrel{\epsilon}{\Longrightarrow} \Delta \vdash c' : \Theta}$$

$$[\textsc{Single}]\ \frac{\Delta \vdash c : \Theta \stackrel{a}{\longrightarrow} \Delta' \vdash c' : \Theta'}{\Delta \vdash c : \Theta \stackrel{a}{\Longrightarrow} \Delta' \vdash c' : \Theta'}$$

$$[\textsc{Seqnc}]\ \frac{\Delta \vdash c : \Theta \stackrel{s}{\Longrightarrow} \Delta' \vdash c' : \Theta' \qquad \Delta' \vdash c' : \Theta' \stackrel{t}{\Longrightarrow} \Delta'' \vdash c'' : \Theta''}{\Delta \vdash c : \Theta \stackrel{st}{\Longrightarrow} \Delta'' \vdash c'' : \Theta''}$$

Table 2.13: *Japl* language: traces

2.5 Traces and the notion of testing

We have seen that the execution of an open program gives rise to a sequence of its interface interactions that occur during the execution. We use these sequences, called traces, to define a semantics of a program which characterizes the program's possible effect on its environment, i.e. on other, external components. Based on this *trace semantics*, we will formalize a notion of testing that builds the formal basis of our testing approach.

Definition 2.5.1 (Trace Semantics)**:** We introduce three semantic functions

$$[\![\cdot]\!]^a_{trace}, [\![\cdot]\!]^p_{trace}, \text{ and } [\![\cdot]\!] : \Delta \vdash p : \Theta \rightharpoonup \mathscr{P}(a^*),$$

such that for well-typed open programs p with $\Delta \vdash p : \Theta$ we define

$$[\![\Delta \vdash p : \Theta]\!]^a_{trace} \stackrel{\text{def}}{=} \{s \in a^* | \Delta \vdash c_{init}(p) : \Theta \stackrel{s}{\Longrightarrow} \Delta' \vdash c' : \Theta'\},$$
$$[\![\Delta \vdash p : \Theta]\!]^p_{trace} \stackrel{\text{def}}{=} \{s \in a^* | \Delta \vdash \overline{c_{init}}(p) : \Theta \stackrel{s}{\Longrightarrow} \Delta' \vdash c' : \Theta'\}, \text{ and}$$
$$[\![\Delta \vdash p : \Theta]\!] \stackrel{\text{def}}{=} [\![\Delta \vdash p : \Theta]\!]^a_{trace} \cup [\![\Delta \vdash p : \Theta]\!]^p_{trace}$$

Thus, the first two semantics definitions yield the set of traces that can be implemented by a program p with some interacting external components, where the program is either considered as a passive or as an active component, respectively. The third definition then consists of all traces that can be realized by the program with any external component.

We want to use the notion of traces to formalize what it means to test a component on the basis of its interface behavior. However, for a better understanding of the differences between a more traditional testing approach and the testing approach we want to embark on, we first will give a definition of a traditional testing approach in the context of our language.

Testing a component, in general, means to execute the component in order to increase confidence in its quality by inspecting the execution. In most cases, a component tester aims at only one feature to be tested at a time. Test cases are

specified each describing a specific execution of the program along with certain expectations related to the feature to be tested. In particular, *traditional* test case specifications basically consist of certain input data and the corresponding anticipated output data. Both, input and output data, may comprise not only values directly communicated between the component and its environment but also the component's states. For this reason, this testing approach is sometimes called *state-based testing* [28]. If the feature to be tested covers only functional requirements, the test case's pass and fail criteria are often completely confined to the specification of the input and output data: the component passes the specified test case if, and only if, its execution with respect to the specified input data leads to the desired output data. Moreover, the expected output data is often not given explicitly but instead a so-called *test oracle* is provided, i.e., a Boolean function of the input and output data determining whether the component passed or failed the test. This testing approach can be formalized in terms of our formal setting as follows:

Definition 2.5.2 (State-based testing)**:** For a given component under test p with $\Theta \vdash p : \Delta$, a state-based test case specification can be represented by

- input data $(h_{in}, \mathsf{v}_{in}, a_c)$ consisting of a heap function h_{in}, a global variable function v_{in}, and an incoming method call label $a_c = \nu(\Theta').\langle call\ o.m(\overline{v_{in}})\rangle?$ as well as of
- a test oracle function $success : (\mathsf{H} \times \mathsf{V} \times a) \to (a \times \mathsf{H} \times \mathsf{V}) \to \mathsf{bool}$ which yields for each input data a Boolean function of the resulting outgoing return label $a_r = \nu(\Delta').\langle return(v_{out})\rangle!$ and the final state of the component.

The component p passes the specified test case if:

$$\Delta \vdash (h_{in}, \mathsf{v}_{in}, \epsilon) : \Theta \stackrel{a_c a_r}{\Longrightarrow} \Delta' \vdash (h_{out}, \mathsf{v}_{out}, \epsilon) : \Theta'$$
$$\text{such that } success(h_{in}, \mathsf{v}_{in}, a_c)(a_r, h_{out}, \mathsf{v}_{out}) \text{ holds.}$$

In order to carry out a state-based test case execution, we have to write a test program which implements a *set-up*, assuring that the component's configuration corresponds to the specified input data, and which furthermore implements the method call a_c as well as the evaluation of the oracle function *success* afterwards.

Unfortunately, as the test program is itself a *Japl* component it can modify neither the global variables of the component under test nor its heap directly, but the test program has to drive the component to the desired input configuration by means of method calls. However, it is often not clear which calls should be made to reach the input configuration or if the configuration can be reached at all. For, it might be that there exists no trace in the component's trace semantics that leads to the desired configuration. Summarizing, the test set-up, initiating a state-based test execution, is difficult or sometimes even impossible to realize in a component-based setting, as the component's state is usually not accessible due to information hiding and encapsulation.

Besides that, according to Definition 2.5.2, state-based testing does not allow for call-backs which might occur between the specified method call and its return. More general, sometimes it might be necessary to specify a test execution which entails a longer communication sequence s in between the original call and its return:

$$\Delta \vdash (h_{in}, \mathsf{v}_{in}, \epsilon) : \Theta \overset{a_c s\, a_r}{\Longrightarrow} \Delta' \vdash (h_{out}, \mathsf{v}_{out}, \epsilon) : \Theta' \ .$$

Even if we relax the criteria for passing the test by allowing the component to produce more interface communication than merely the return label, then this requires at least additional declarations of the reaction that has to be carried out by the test program.

In a testing approach that is based on the component's interface behavior, in contrast, the component's internal state is not specified but only its behavior which is observable by a test program. This approach is therefore called *behavior-based testing*. In our setting, the observable behavior consists of method calls and returns occurring at the component's interface, thus, a test case specification stipulates a sequence of interface interactions.

Definition 2.5.3 (Behavior-based testing)**:** For a given component under test p with $\Theta \vdash p : \Delta$ a *behavior-based test case specification* can be represented by a sequence of communication labels, that is, a trace $s \in a^*$.

The component p passes the specified test case if:

$$\Delta \vdash c_{init}(p) : \Theta \overset{s}{\Longrightarrow} \Delta' \vdash c : \Theta'$$
$$\text{or } \Delta \vdash \overline{c_{init}}(p) : \Theta \overset{s}{\Longrightarrow} \Delta' \vdash c : \Theta' \ .$$

In other words, we actually test for $s \in [\![\Delta \vdash p : \Theta]\!]$. Note that this approach also allows for test cases which are initiated by the component under test, that is, we also can execute and test the component's main body. A test trace s generally describes communication steps which are expected to be carried out by the component under test as well as communication steps that have to be carried out by the component's environment. Thus, when conducting behavior-based testing, a test program need not to take care for establishing a specified input configuration of the component under test but in exchange it has to implement several calls and returns along the test execution as stipulated by the specified test trace. At the same time, the test program has to check that the component's contribution to the interface communication complies with the trace specification.

Specifically, in order to test for $s \in [\![\Delta \vdash p : \Theta]\!]$ for a component p and a test trace s, we need a test program which can implement the *complementary trace* $\bar{s}$ of s that results from s by replacing question marks with exclamation marks, and vice versa; hence, we require a test program p_{tp} with $\Theta \vdash p_{tp} : \Delta$ and $\bar{s} \in [\![\Theta \vdash p_{tp} : \Delta]\!]$. Then, a test execution can be considered as the execution of actually both programs p and p_{tp}, such that p_{pt}'s outgoing communication represents the incoming communication of p and vice versa such that p_{pt} reports a successful test run if it observes trace $\bar{s}$ at its interface. One question that may

arise from this testing approach is how to find an appropriate test program for a given test case specification. In fact, the following two chapters will deal with the introduction of, both, a test specification language that is built on the basis of interface traces and a test program generation algorithm.

But first it remains to show, that our language represents a sound framework for further investigations. More precisely, the semantics' extension by interface communication rules should represent a sound extension with regards to the semantics of the internal computation. In context of the above mentioned test approach, this means that, under the provision that the program p passes the test, the two programs p and p_{pt} can be merged to a single program that gives rise to a sequence of internal computation steps containing internal call and return steps that correspond to the labels of the trace s. This is more general formalized in the following lemma.

Definition 2.5.4 (Merge of components)**:** Let

$$p_1 = \overline{impdecl_1};\ \overline{T_1}\ \overline{x_1};\ \overline{cldef_1}\ \{\overline{T_1'}\ \overline{x_1'};\ stmt_1;\ \texttt{return}\}$$
and
$$p_2 = \overline{impdecl_2};\ \overline{T_2}\ \overline{x_2};\ \overline{cldef_2}\ \{\overline{T_2'}\ \overline{x_2'};\ stmt_2;\ \texttt{return}\}$$

be two components such that $\Theta_2 \vdash p_1 : \Theta_1$ and $\Theta_1 \vdash p_2 : \Theta_2$. Then the *merge of the two components* is defined as follows,

$$p_1 \Cup p_2 \stackrel{\text{def}}{=} \overline{cldef_1}\ \overline{cldef_2}\ \overline{T_1}\ \overline{x_1};\ \overline{T_2}\ \overline{x_2};\ \{\overline{T_1'}\ \overline{x_1'};\ stmt_1;\ \texttt{return}\},$$

where we assume an ad-hoc renaming of global variables to prevent name clashes if necessary.

Note that the merge p of the two components p_1 and p_2 does not contain their mutual import declarations and only the main body of p_1. In particular, the merge is therefore not symmetric. The intuitive motivation for the drop of one main body is that one component represents the ("main") program and the other one is incorporated as some kind of a passive library. For, we have seen in the operational semantics that always exactly one main body is executed.

Lemma 2.5.5 (Compositionality)**:** There exists a merge function on configurations

$$\cdot \Cup \cdot : Conf \times Conf \rightharpoonup Conf$$

with the following two properties:

1. For two components p_1 and p_2 with $\Theta_2 \vdash p_1 : \Theta_1$ and $\Theta_1 \vdash p_2 : \Theta_2$, such that

$$\Theta_2 \vdash c_{init}(p_1) : \Theta_1 \stackrel{t}{\Longrightarrow}_{p1} \Theta_2' \vdash c_1' : \Theta_1'$$
and
$$\Theta_1 \vdash \overline{c_{init}}(p_2) : \Theta_2 \stackrel{\bar{t}}{\Longrightarrow}_{p2} \Theta_1' \vdash c_2' : \Theta_2' ,$$

and for $p = p_1 \Cup p_2$ such that $\vdash p : \Theta_1, \Theta_2$ the following holds:

$$c_{init}(p) \longrightarrow_p^* c_1' \Cup c_2' ,$$

We annotated the transition arrows to indicate the context in which the respective transition rules are applied.

2. Assume a closed program p with $\vdash p : \Theta$. For every two components p_1 and p_2 with $\Theta_2 \vdash p_1 : \Theta_1$ and $\Theta_1 \vdash p_2 : \Theta_2$ such that $p_1 \Join p_2 = p$ and $\Theta_1, \Theta_2 = \Theta$ the following holds:

   $c_{init}(p) \longrightarrow_p^* c$ implies the existence of a trace $t \in a^*$ such that:

   - $\Theta_2 \vdash c_{init}(p_1) : \Theta_1 \stackrel{t}{\Longrightarrow}_{p1} \Theta_2' \vdash c_1' : \Theta_1'$,
   - $\Theta_1 \vdash \overline{c_{init}}(p_2) : \Theta_2 \stackrel{\bar{t}}{\Longrightarrow}_{p2} \Theta_1' \vdash c_2' : \Theta_2'$ and
   - and $c_1' \Join c_2' = c$.

In words: Feeding a component p_1 with the interface communication carried out by p_2, and vice versa, has the same effect as syntactically merging the two programs, such that the communication is carried out internally. On the other hand, a closed program can be torn into two components which can communicate via interface communication reaching configurations that correspond to the configurations reachable by the original component.