DOI 10.1007/s00165-012-0259-y
BCS © 2012
Formal Aspects of Computing (2014) 26: 491–535

Formal Aspects of Computing

An operational semantics for object-oriented concepts based on the class hierarchy

Robert J. Colvin

The University of Queensland, Queensland Brain Institute, St Lucia, QLD 4072, Australia

Abstract. The formalisation of object-oriented languages is essential for describing the implementation details of specific programming languages or for developing program verification techniques. However there has been relatively little formalisation work aimed at abstractly describing the fundamental concepts of object-oriented programming, separate from specific language considerations or suitability for a particular verification style. In this paper we address this issue by formalising a language that includes the core object-oriented programming language concepts of field tests and updates, methods, constructors, subclassing, multithreading, and synchronisation, built on top of standard sequential programming constructs. The abstract syntax is relatively close to the core of typical object-oriented programming languages such as Java. A novel aspect of the syntax is that objects and classes are encapsulated within a single syntactic term, including their fields and methods. Furthermore, class terms are structured according to the class hierarchy, and objects appear as subterms of their class (and method instances as subterms of the relevant object). This helps to narrow the gap between how a programmer thinks about their code and the underlying mathematical objects in the semantics. The semantics is defined operationally, so that all actions a program may take, such as testing or setting local variables and fields, or invoking methods on other objects, appear on the labels of the transitions. A process-algebraic style of interprocess communication is used for object and class interactions. A benefit of this label-based approach to the semantics is that a separation of concerns can be made when defining the rules of the different constructs, and the rules tend to be more concise. The basic rules for individual commands may be composed into more powerful rules that operate at the level of classes and objects. The traces generated by the operational semantics are used as the basis for establishing equivalence between classes.

Keywords: Operational semantics; Object-oriented programming; Process algebra; Class hierarchy

1. Introduction

Object-oriented programming [KLW95, GHJV95, Mey00] is a widely used programming paradigm, as can be observed by the popularity of languages such as Java [AGH00] and C++ [Str97]. The paradigm originated from ideas in the 1960s, in particular the programming language Simula [BDMN73], and developed later in Smalltalk [GR02]. The style lends itself to encapsulation and code reuse, both important features for writing structured code; it is also a structure that can be beneficial for formal specification [Smi00] and modelling biological systems [FHN+11].

Correspondence and offprint requests to: R. J. Colvin, E-mail: r.colvin@uq.edu.au

Object-oriented programs are structured by grouping related functionality into classes, and can be large and complex and used for large-scale or safety-critical applications. To take advantage of modern multi-processor computer architectures, many also provide the ability to create concurrent processes (threads). Due to their importance and complexity, analysis of such languages is an active research topic [DGC95, AF99, HM01, Nip03, LLM07], including the complete, formal definition and analysis of the languages Java and C# by Börger, Stark, Fruja et al. [SSB01, BFGS05, Fru04, Stä05, Fru10].

The semantics provided in the literature are typically designed towards the goal of formal analysis, and it is therefore often the case that the semantics are either language-specific or omit important details (such as inheritance or multithreading). There has been relatively little work on describing the fundamental concepts of object-oriented programming, in which classes, objects, and methods are active (concurrent) participants. The contribution of this paper is in providing a formalisation of an object oriented language whereby high-level concepts such as dynamic dispatch are described with reference to the class hierarchy, and classes, objects, and methods are encapsulated processes that operate and interact with each other. This helps to narrow the gap between how a programmer or student may conceive of the semantics of object-oriented concepts, and the mathematical constructs in the formalisation.

The language has a familiar sequential code base (including exceptions), which is extended by object-oriented features such as fields, methods, constructors, and subclassing. A scheme for multithreading (concurrency) and ensuring mutual exclusion on data is also presented. The language is described incrementally (following the work of Börger et al.), and the construct-specific rules may be composed to define powerful structural rules that operate at the class or object level.

The formal semantics is defined operationally, that is, it is based on transitions that a language construct can take. This approach was popularised by Plotkin [Plo81] (see also [Plo04]) and his treatment of sequential programming languages with procedures and recursion. The approach we use in this paper is different from Plotkin's in that the type of the step taken is represented syntactically in the label on the transition. The context in which a command is executed determines whether that step is allowed, and the context may change as a result. The rules tend to be concise, because only contextual information relevant to the particular language construct need appear. The method call mechanism (and all other inter-process communication) is based on that of CCS [Mil82].

We take the approach that the syntax of the language should match as closely as possible with actual programming languages, with the semantic framework appearing mostly in the labels on transitions; however, in some cases, abstract commands appear within the syntax of programs, but these are added and removed automatically during execution. We do not consider static semantics issues such as type correctness (although some errors can be caught dynamically by exceptions), private and public fields, interfaces, etc. Because our language is similar to concrete programming languages such as Java, existing type-checking techniques (e.g., [SSB01, KN06]) can be employed to ensure well-formedness.

The paper is structured as follows: in Sect. 2 we first present an overview of the syntax and semantics, focusing on class- and object-level behaviour of the system. In Sect. 3 we present the syntax and semantics of a simple sequential programming language with exceptions, and abstract input/output. In Sect. 4 we introduce the inter-process communication framework that is used throughout the paper. In Sect. 5 we give the semantics of testing and updating fields. In Sect. 6 we describe the semantics of methods, and in Sect. 7 we describe how new objects may be constructed. In Sect. 8 we describe how subclassing is handled, and in Sect. 9 we present a scheme (based on Java) for introducing thread-based concurrency and synchronisation. In Sect. 10 we explore tool support and define a method for showing equivalence between classes. In Sect. 11 we compare with related work.

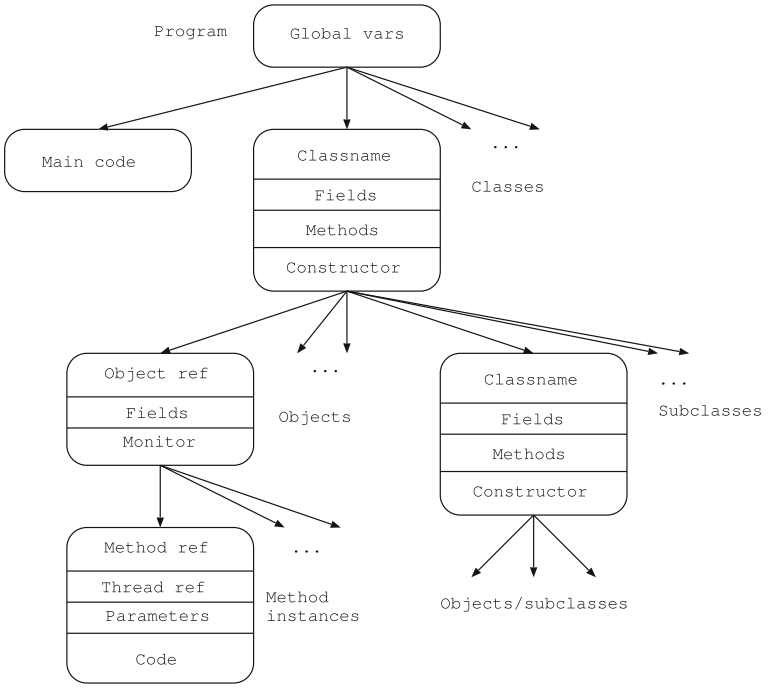

Fig. 1. Hierarchical structure of processes in an object-oriented program

2. Overview

2.1. Program structure and syntax

Our approach to defining the semantics of an object-oriented language is based on interacting *processes*, where a process contains relevant subprocesses, e.g., a process representing a class contains all objects of that class as subprocesses. As such, the syntax of the language matches as closely as possible the *class hierarchy*, with subclasses nested inside their parent class. This is extended further by nesting objects of a class inside that class, with method instances nested inside the relevant object. A graphical representation of the general structure is given in Fig. 1. At the top level the program may declare variables, which are both syntactically and semantically global in scope. The program consists of some *Main* code, which is used to start the program, and a set of classes. Each class may declare its own class-level fields and class- and object-level methods,[1] as well as a constructor for creating objects. A class contains all objects of that class, and may contain other classes (its subclasses). Objects have their own fields, as well as other information such as a monitor for restricting access to its state. Each object contains a method instance for each method invocation to which it is responding.

A complete abstract syntax for the language is given in Fig. 2. We assume the type *Ident* for variable, field, and method identifiers, *Val* of basic values, the type *Expr* of typical expressions, including method calls and constructor calls, the type *Cmd* for commands, and the type *Exception* for exceptions. Object, class, and thread references are taken from the type *Ref*. A state σ is a partial mapping from identifiers to values, and a method definition ρ is a partial mapping from (method) identifiers to functions that return a command instantiated with a sequence of actual parameters. A constructor γ returns a new object with the object reference determined by the parameter.

The basic statements (*Std*) include those standard in sequential programming languages, as well as abstract input/output commands, exceptions, and object oriented concepts such as field references, method calls, constructor calls, and synchronisation. There are four declaration commands (*Decl*), which declare and define the scope of variables (global and local variables), (class and object) fields, (class and object) methods, and constructors.

[1] In Java, class-level fields and methods are declared with the keyword `static`, and we will adopt this convention when giving concrete syntax.

Assume:

$$x, f, m \in \mathit{Ident} \quad e, b \in \mathit{Expr} \quad c \in \mathit{Cmd} \quad \mathit{exc} \in \mathit{Exception} \quad o, \mathrm{CL}, \mathrm{T} \in \mathit{Ref} \quad i, n \in \mathbb{N}$$
$$\sigma \in \mathit{Ident} \nrightarrow \mathit{Val} \qquad \rho \in (\mathit{Ident} \nrightarrow (\mathrm{seq}\ \mathit{Val} \nrightarrow \mathit{Cmd})) \qquad \gamma \in \mathbb{N} \rightarrow \mathit{Cmd}$$

	Cmd/Expr	*Rules*
Std	**nil**	–
	$x := e$	10
	$c_1 \ ; \ c_2$	11
	if b **then** c_1 **else** c_2	12
	while b **do** c	13
	throw exc	20
	try c_1 **catch** $(\mathit{exc})\ c_2$	22
	input	23
	output e	24
	$e.f$	30
	$e_1.f := e_2$	31
	$m(\vec{e}),\ e.m(\vec{e})$	35, 55
	return	36
	new $\mathrm{CL}(\vec{e})$	42
	$e.\mathrm{CL}{:}\ m(\vec{v})$	45
	e **instOf** CL	48
	$(e_1)\ e_2$	$(\mathit{defined})$
	sync $e \bullet c$	58

	Cmd/Expr	*Rules*
Decl	**state** $\sigma \bullet c$	$14-17, 19$
	fields $\sigma \bullet c$	29, 32, 33
	methods $\rho \bullet c$	39, 56
	cstr $\gamma \bullet c$	44
Ref	**ref** $o \bullet c$	34, 47, 49
	$\mathbf{mi}_i\ c$	$40-41$
	$\mathbf{thr}_{\mathrm{T}}\ c$	53, 54
Abs	**Program** c	25
	$c_1 \parallel c_2$	26, 27
	wait o_i	37
	$\mathbf{wksp}_i\ c$	38, 43, 46
	release	50
	newtref c	51
	$\mathbf{trefs}_i\ c$	52
	mtr $(\mathrm{T}, n) \bullet c$	57
	$\lock o \bullet c$	58, 59

Fig. 2. Language summary

$$\begin{aligned}
(\mathbf{obj}\ o(\sigma)_i \bullet c) &\mathrel{\widehat{=}} (\mathbf{ref}\ o \bullet \mathbf{fields}\ \sigma[\mathbf{this} \mapsto o] \bullet \mathbf{wksp}_i\ c) \\
(\mathbf{class}\ \mathrm{CL}(\sigma, \rho, \gamma)_j \bullet c) &\mathrel{\widehat{=}} (\mathbf{ref}\ \mathrm{CL} \bullet \mathbf{fields}\ \sigma \bullet \mathbf{methods}\ \rho \bullet \mathbf{cstr}\ \gamma \bullet \mathbf{wksp}_j\ c) \\
\gamma &\mathrel{\widehat{=}} \lambda\, n \bullet (\mathbf{obj}\ \mathrm{CL}_n(\sigma_{\mathit{default}})_0 \bullet \mathbf{nil}) \\
\mathcal{P} &\mathrel{\widehat{=}} \mathbf{Program}\ (\mathbf{state}\ \sigma_{\mathit{globals}} \bullet \mathit{Main} \parallel \mathcal{C}_1 \parallel \ldots \parallel \mathcal{C}_n)
\end{aligned} \tag{1}$$

Fig. 3. Standard templates

There are three reference types (*Ref*), for distinguishing and scoping objects and classes, method instances, and giving the thread reference of a method instance. Abstract commands (*Abs*), which do not appear in concrete syntax explicitly (but may be implied), include the top-level delimiter **Program**, parallel composition, **wait** for waiting for a method call to return, a workspace where an object or class keeps its methods and objects, and commands related to multithreading.

Processes such as classes, objects and method instances have unique references within the system: class names are user-generated; object references are class names subscripted by natural numbers; and method instance references are object references further subscripted by natural numbers. A method instance may also have a thread reference.

Figure 3 introduces abbreviations for the standard structure of processes corresponding to objects and classes, constructors, and a top-level program. An object is formed from a reference, a set of fields which includes the special field **this**, and a workspace which may contain method instances. A class is formed from a reference (the class name), a set of class-level (`static`) fields, method definitions, a constructor, and a workspace which may contain objects of that class, subclasses, and/or class-level method instances. A constructor call $\gamma(n)$ returns an object with reference CL_n (i.e., the class name subscripted by a number unique within that class), default values for its fields, and an empty workspace. A program $\mathcal{P}$ is typically formed from a set of global variables $\sigma_{\mathit{globals}}$, and the parallel composition of some *Main* process (sequential code), and n classes.

Figure 4 gives an example program written in abstract syntax. At the topmost level the program contains a global variable, g, which is declared with the **state** keyword, and is initially 0. The *Main* process is some sequential code that has local variable r, initially null, which is assigned the reference to a new *Rectangle* object of dimensions 2×1. The object is then scaled by a factor determined by user input (the parameter to the method call is the abstract expression **input**). It thereafter proceeds by executing some command c, which we leave unspecified.

$$
\begin{array}{lr}
\textbf{Program}\quad \textbf{state}\ \{g \mapsto 0\} \bullet & \textit{Global state} \\
\quad \textbf{state}\ \{r \mapsto \mathsf{null}\} \bullet r := \textbf{new}\ \textsc{rect}(2,1)\ ;\ r.\mathit{scale}(\textbf{input})\ ;\ c & \textit{Main} \\
\quad \| & \\
\quad \textbf{class}\ \textsc{rect}(\{\mathit{tarea} \mapsto 0\}, \{\mathit{init} \mapsto .., \mathit{getHt} \mapsto .., \mathit{scale} \mapsto ..\}, \gamma_{\textsc{re}})_1 \bullet & \textit{Rectangle} \\
\qquad \textbf{class}\ \textsc{sq}(\varnothing, \{\mathit{init} \mapsto .., \mathit{getSide} \mapsto ..\}, \gamma_{\textsc{sq}})_1 \bullet \textbf{nil} & \textit{Square}
\end{array}
$$

Fig. 4. An example program with main code and two classes

In parallel with the *Main* process is a process for a class defining a *Rectangle*. It has class name RECT, a class-level field *tarea*, which maintains the current total area of all *Rectangle* objects, and defines three methods, *init*, *getHt* and *scale*. The method *init* is an *initialiser*, and is called only when a new object is constructed. The function $\gamma_{\textsc{re}}$ is the constructor for *Rectangle* objects. The class has a *workspace* (**wksp**), which maintains a parallel composition of subprocesses, and a count which strictly increases as *Rectangle* objects are created and is used to generate unique object and method instance references. Initially, the count is 1, and there are no *Rectangle* objects. However, the workspace does contain the process *Square*, which is a subclass of *Rectangle*. It has no new fields, but does declare a new method, *getSide*, overrides the initialiser *init*, and has its own constructor, $\gamma_{\textsc{sq}}$. Its workspace is **nil** (no *Square* objects and no further subclasses).

A process $\mathcal{O}$ corresponding to a *Rectangle* object of dimensions 2×1 is as follows.

$$\mathcal{O} \mathrel{\widehat{=}} \textbf{ref}\ \textsc{rect}_1 \bullet \textbf{fields}\ \{h \mapsto 2, w \mapsto 1\} \bullet \textbf{wksp}_1\ \textbf{nil}$$

The reference is the *Rectangle* class name with a natural number subscript, $\textsc{rect}_1$. Further *Rectangle* objects will be called $\textsc{rect}_2$, $\textsc{rect}_3$, etc. Such objects appear in the workspace of the *Rectangle* process. The field names h and w do not need to be distinguished from the same variables in other *Rectangle* objects; the scopes are separate and this is reflected in the semantics.

Throughout the paper, with the exception of the generic object reference o, we follow the convention of writing reference values in SMALL CAPS font, e.g., RECT for the *Rectangle* class name. Object references are subscripted versions of their class name, e.g., $\textsc{rect}_1$ above. This naming strategy makes it easy to ensure objects are uniquely identified by their reference. (However, any naming scheme that ensures uniqueness may be used.) For brevity, we write R as a shorthand for $\textsc{rect}_1$. Method references are subscripted versions of the object reference, i.e., the ith method instance of the jth *Rectangle* object has the reference $\textsc{rect}_{j,i}$. Such method references will typically appear in the more compact form o_i or $\textsc{r}_i$.

Objects and classes may respond to method calls, and this creates a new instance of the relevant method, which we call method instances (also known as stack frames), in the workspace of the relevant process. The standard form of a method instance is a method number, unique within that object, and some sequential code. For instance, the following process $\mathcal{M}$ corresponds to a method instance of *getHt* on object $\mathcal{O}$.

$$\mathcal{M} \mathrel{\widehat{=}} \textbf{mi}_1\ \textbf{return}\ h$$

The method number is 1, and is combined with the enclosing object reference to give a system-wide unique method reference, e.g., $\textsc{r}_1$.

2.2. Semantics

The semantics is defined operationally; a set of rules define how a program is executed. The semantics is 'small-step', specifying the evaluation of expressions and execution of commands at a fine-grained level—for instance, the evaluation of expression '$x + 1$' is a two-step process, where first the value of x is found, and then one is added to give the result. In a concurrent system, parallel processes may interleave their steps during the evaluation.

In contrast to most operational semantics, in this paper each step is represented by a label, which is a syntactic abstraction of the kind of step being taken. Labels are promoted through the syntax of the program, which, as described above, specifies the local context. As labels are promoted, they may be modified themselves as well potentially modify the context. Labels are also used to represent communications between processes (which may or may not be dependent on the context).

$$\textit{Assume} \quad co, cc, cm \in \mathit{Cmd}$$

$$\begin{array}{rclrcl} \mathcal{O} & = & (\mathbf{obj}\ o(\sigma)_i \bullet co) & \mathcal{C} & = & (\mathbf{class}\ \mathrm{CL}(\sigma_{\mathrm{CL}}, \rho, \gamma)_j \bullet cc) \\ \mathcal{O}[\mathcal{M}] & = & (\mathbf{obj}\ o(\sigma)_i \bullet co \parallel \mathcal{M}) & \mathcal{C}[\mathcal{O}] & = & (\mathbf{class}\ \mathrm{CL}(\sigma_{\mathrm{CL}}, \rho, \gamma)_j \bullet cc \parallel \mathcal{O}) \\ \mathcal{O}^+\mathcal{M} & = & (\mathbf{obj}\ o(\sigma)_{i+1} \bullet co \parallel \mathcal{M}) & \mathcal{C}^+\mathcal{O} & = & (\mathbf{class}\ \mathrm{CL}(\sigma_{\mathrm{CL}}, \rho, \gamma)_{j+1} \bullet cc \parallel \mathcal{O}) \\ \mathcal{M} & = & (\mathbf{mi}_k\ cm) & & & \end{array} \tag{2}$$

Rule 1 (Method call (invoke)).

$$o.m(\vec{v}) \xrightarrow{o_i.m(\vec{v})!} \mathbf{wait}\ o_i \xrightarrow{o_i.\mathsf{return}?} \mathbf{nil}$$

Rule 2 (New object (invoke)).

$$\frac{o = \mathrm{CL}_j}{\mathbf{new}\ \mathrm{CL}(\vec{v}) \xrightarrow{\mathsf{new}\ o!} o.init(\vec{v})}$$

Rule 3 (Method call (respond)).

$$\frac{m \in \mathrm{dom}(\rho) \qquad \mathcal{M} = (\mathbf{mi}_i\ \ m_\rho(\vec{v}))}{\mathcal{C}[\mathcal{O}] \xrightarrow{o_i.m(\vec{v})?} \mathcal{C}[\mathcal{O}^+\mathcal{M}]}$$

Rule 4 (New object (respond)).

$$\frac{\mathcal{O} = \gamma(j)}{\mathcal{C} \xrightarrow{\mathsf{new}\ \mathrm{CL}_j\ ?} \mathcal{C}^+\mathcal{O}}$$

Rule 5 (Overridden method call (respond)).

$$\frac{m \in \mathrm{dom}(\rho_2) \quad \mathcal{M} = (\mathbf{mi}_i\ \ m_{\rho_2}(\vec{v}))}{\mathcal{C}_1[\mathcal{C}_2[\mathcal{O}]] \xrightarrow{o_i.m(\vec{v})?} \mathcal{C}_1[\mathcal{C}_2[\mathcal{O}^+\mathcal{M}]]}$$

Rule 6 (Super method call (respond)).

$$\frac{m \in \mathrm{dom}(\rho_1) \setminus \mathrm{dom}(\rho_2) \quad \mathcal{M} = (\mathbf{mi}_i\ \ m_{\rho_1}(\vec{v}))}{\mathcal{C}_1[\mathcal{C}_2[\mathcal{O}]] \xrightarrow{o_i.m(\vec{v})?} \mathcal{C}_1[\mathcal{C}_2[\mathcal{O}^+\mathcal{M}]]}$$

Rule 7 (Communication).

$$\frac{c_1 \xrightarrow{\ell!} c_1' \qquad c_2 \xrightarrow{\ell?} c_2'}{c_1 \parallel c_2 \xrightarrow{\tau} c_1 \parallel c_2'}$$

Fig. 5. Derived structural rules

This style of operational semantics, where labels are promoted and either operate on the local context, or represent a communication, or both, complements the structured nature of the syntax and supports encapsulation of local data and straightforward communication between parallel processes. Each command given above in Fig. 2 is specified by rules showing which labels it generates or which labels it interacts with; other labels are simply promoted unchanged. This style supports modularity as defined by Mosses [Mos04b]. The advantage of this style of operational semantics and structuring of syntax manifests in the derived rules in Fig. 5, described below, which operate at the level of objects and classes as defined by the templates in Fig. 3. These rules are relatively concise, with the syntactic structure of the relevant processes implicitly conveying some of the intent of the rule. In similar rules for method invocation and object construction in the literature, there are a significant number of antecedents to the rules and a large syntactic overhead.

As an example of labels modifying local context or communicating to modify some remote context, consider an assignment statement $x := 0$. This transitions to **nil** with a label indicating the update, i.e., $x := 0 \xrightarrow{x := 0} \mathbf{nil}$. This label is promoted through the structure of the enclosing program, which may include x as a field; the value of field x is modified to be 0, and the label itself is modified to become silent (τ), so that it will have no effect on any other context. If instead of updating a local variable, a field of some other object is updated, then the label is decorated with an exclamation mark '!' to indicate it is an *invocation*, that is, a request of or directive to another object, e.g., $o.x := 0 \xrightarrow{o.x := 0!} \mathbf{nil}$. The corresponding *response* transition is given by $\mathcal{O} \xrightarrow{o.x := 0?} \mathcal{O}'$, where $\mathcal{O}$ is some parallel process representing the target object which contains a local declaration of x, and $\mathcal{O}'$ is identical to $\mathcal{O}$ except for the updated value of field x. The labels are said to *match*, and synchronise using a communication style based on CCS [Mil82], as exemplified by Rule 7. The steps of the invoker and responder combine into a single, internal step.

As an example of how labels are promoted through nested contexts, consider the command c defined below.

$$c \mathrel{\widehat{=}} (\textbf{if } x \textbf{ then } h := 1 \textbf{ else } h := -1)\ ;\ x := \mathit{false}$$

There are two possible behaviours (*traces*) of the program, corresponding to the initial boolean value of x. Traces are formed from multiple applications of the operational rules, but below we summarise the effect in a single bundled transition, that is, the notation $c \stackrel{\ell s}{\Longrightarrow} c'$ represents a sequence of transitions with labels given by the sequence ℓs (elements separated by whitespace). By convention we omit internal steps from traces.

$$c \xrightarrow{x=\mathit{true}\quad h:=1\quad x:=\mathit{false}} \textbf{nil} \qquad\qquad c \xrightarrow{x=\mathit{false}\quad h:=-1\quad x:=\mathit{false}} \textbf{nil}$$

When executed inside a local state that records x as currently true, those parts of the label referring to x become internal steps (τ), and updates to x are recorded by its changing value in the local state. The second trace above is not consistent with a value of *true* for x, and is thus eliminated (or pruned) from the possible behaviours.

$$(\textbf{state } \{x \mapsto \mathit{true}\} \bullet c) \stackrel{h:=1}{\Longrightarrow} (\textbf{state } \{x \mapsto \mathit{false}\} \bullet \textbf{nil})$$

The labels referencing x have become internal steps (and omitted from the transition arrow). At an outer level that declares h, the remaining label is also hidden.

$$(\textbf{state } \{h \mapsto 0\} \bullet (\textbf{state } \{x \mapsto \mathit{true}\} \bullet c)) \Longrightarrow (\textbf{state } \{h \mapsto 1\} \bullet (\textbf{state } \{x \mapsto \mathit{false}\} \bullet \textbf{nil}))$$

This situation of nested states corresponds with a local variable declaration inside a method instance (x) occurring inside the scope of an object-level field (h). The operations of a method body (such as c) on a method-level variable, or object-level or class-level field, are hidden outside the scope of the method, object or class. This serves to keep the semantic scope of a variable equivalent to its syntactic scope, and implicitly handles multiple instance of the same variable or field name occurring in different methods or objects.

At the top of Fig. 5 we abbreviate an object of the standard structure to $\mathcal{O}$ and class to $\mathcal{C}$. We also let $\mathcal{O}[\mathcal{M}]$ abbreviate an object that is executing method instance $\mathcal{M}$ in its workspace, and analogously $\mathcal{C}[\mathcal{O}]$ picks out a particular object in the workspace of class $\mathcal{C}$. The notation $\mathcal{O}^+\mathcal{M}$ is object $\mathcal{O}$ extended to include a new method instance $\mathcal{M}$ (the difference with $\mathcal{O}$ being that the count i used to generate unique labels is incremented and $\mathcal{M}$ is added to the workspace). Analogously, $\mathcal{C}^+\mathcal{O}$ is the result of a new object of class $\mathcal{C}$ being constructed. A method instance $\mathcal{M}$ is just some piece of sequential code cm wrapped inside a method reference k. In the rules in the figure, the processes are assumed to be defined according to those standard forms unless otherwise specified in the antecedents of the rule.

Let us first consider invocation of and then response to a method call. The standard command $o.m(\vec{v})$ invokes method m on object o with the list of (evaluated) parameters $\vec{v}$. (To avoid distraction in this section we only consider commands where expressions have already been fully evaluated.) The actual parameters may include values of regular types (numbers, booleans, etc.), and also references to objects (elements of *Ref*), which can then be referenced by the callee. After calling the method the invoker is suspended until the callee returns (possibly with a value). This is shown by Rule 1, which combines two steps. The syntax of the label on the first step is essentially the same as the command syntax, with the addition of the subscript i which serves to disambiguate this call from any other call of $o.m$ that may be active in the system. The combined reference o_i is used to uniquely identify the relevant labels. The suspended invoker is given by the command **wait** o_i, which may proceed only by responding to a return command from the callee, after which the command terminates.

Rule 3 is the corresponding rule for responding to a method call. An object $\mathcal{O}$ of class $\mathcal{C}$, the target of the invocation, extends its workspace with a new method instance $\mathcal{M}$, provided m is defined in class $\mathcal{C}$ and $\mathcal{M}$ is the instantiation of m with the given parameters $\vec{v}$. (The notation $m_\rho(\vec{v})$ is a shorthand for $\rho(m)(\vec{v})$, that is, extracting the definition of m from ρ and instantiating with actual parameters $\vec{v}$).

When a method instance terminates, it returns control, or control and a value, to the invoker. This gives the following characteristic behaviour of a class and object responding to an invocation.

$$\mathcal{C}[\mathcal{O}] \xrightarrow{o_i.m(\vec{v})?} \mathcal{C}[\mathcal{O}^+\mathcal{M}] \Longrightarrow \mathcal{C}'[\mathcal{O}'^+(\mathbf{mi}_i\ \ \textbf{nil})] \xrightarrow{o_i.\mathsf{return}!} \mathcal{C}'[\mathcal{O}'] \tag{3}$$

The trace is bookended by the complementary labels in Rule 1. The middle transition contains the steps taken in the execution of $\mathcal{M}$, which may involve further method calls, modification of fields, etc. When the execution of the body of $\mathcal{M}$ terminates, a return label is invoked, triggering the caller to continue execution, and the method instance is removed from the workspace of $\mathcal{O}$. Processes $\mathcal{C}'$ and $\mathcal{O}'$ are structurally the same as $\mathcal{C}$ and $\mathcal{O}$, but with the values of their fields modified according to $\mathcal{M}$.

Rule 3 holds for the straightforward case where $\mathcal{O}$ is of class $\mathcal{C}$ and $\mathcal{C}$ defines the method in question. However we may also specify what happens in the presence of inheritance (subclassing). If $\mathcal{C}_2$ is a subclass of $\mathcal{C}_1$, and $\mathcal{O}$ is of class $\mathcal{C}_2$, then it uses the definition of m given by $\mathcal{C}_2$, if it exists (Rule 5). However, if $\mathcal{C}_2$ does not define m, then $\mathcal{O}$ uses the definition of m in $\mathcal{C}_1$, provided it exists (Rule 6). These rules give the semantics of *dynamic dispatch*.

Let us now consider creating a new object. An invocation of a constructor is a two-step process. Firstly, a new default object is created in the workspace of the relevant class, and then an *initialiser* is invoked on that new object, corresponding to the code for that constructor. This is given by Rule 2, which uses o as an abbreviation for CL_j, the new unique reference for the object. The initialiser is just a method that uses the reserved name $init$, and is invoked according to Rule 1 for method calls (to avoid distraction, in this section we assume each class has exactly one initialiser, but account for multiple initialisers in the body of the paper). When it finishes executing, the initialiser returns the reference of the new object to the caller.

Rule 4 summarises the response of the target class. The class $\mathcal{C}$ contains a constructor, γ, which is a function that generates a new default object, with unique reference determined by parameter j. A new object is of the standard form ($\mathcal{O}$), with default values of the fields in σ and where the workspace (co) is **nil**. This new object may now respond to method calls, and in particular, to the initialiser invocation which always immediately follows.

We can summarise the effect of object creation, prior to executing the initialiser, by the following derived rule. Assume $o = \text{CL}_j$.

$$\frac{\mathcal{O} = \gamma(j) \quad \mathcal{M} = (\mathbf{mi}_0 \;\; init_\rho(\vec{v}))}{\mathcal{C} \xrightarrow{\text{new}(o)? \quad o_0.init(\vec{v})?} \mathcal{C}^+\mathcal{O}^+\mathcal{M}} \tag{4}$$

This rule states that class $\mathcal{C}$ responding to the invocation of a new object and the corresponding initialiser is extended by the creation of a new object $\mathcal{O}$ in its workspace, which is ready to execute method instance $\mathcal{M}$. The new object is generated by $\mathcal{C}$'s constructor, and the method instance is an instantiation of the initialiser $init$. The behaviour continues in the general form given in (3).

The behaviour of invokers and responders operating in parallel is given by Rule 7. The labels must syntactically match exactly (i.e., specify the same object, class, parameters, method instance references, etc.), and then each process concurrently takes a step and the label becomes silent.

Let us now consider using the rules to give the execution of the program in Fig. 4, starting with the $Main$ process. In the steps below we let R abbreviate the object reference RECT_1.

$$(\mathbf{state}\ \{r \mapsto \mathsf{null}\} \bullet r := \mathbf{new}\ \text{RECT}(2,1)\ ;\ r.scale(\mathbf{input})\ ;\ c)$$

$$\xrightarrow{\text{new(R)!} \quad \text{R}_0.init(2,1)! \quad \text{R}_0.\text{return?}}$$ Rule 2; Rule 1.

$$(\mathbf{state}\ \{r \mapsto \text{R}\} \bullet r.scale(\mathbf{input})\ ;\ c)$$

$$\xrightarrow{\text{input(3)} \quad \text{R}_1.scale(3)! \quad \text{R}_1.\text{return?}}$$ Receive input from the environment; Rule 1.

$$(\mathbf{state}\ \{r \mapsto \text{R}\} \bullet c)$$

In the first transition a new rectangle object is created and then the initialiser is invoked. This has the effect of updating the value of local variable r to the object reference R. In the second transition the value 3 is received from the environment and this is used as the parameter to the call to scale. Once it has returned the $Main$ process may continue executing c. Note that in the traces the labels associated with the invocation of the initialiser have the subscript 0, and those associated with the invocation of $scale$ have the subscript 1. This strictly ascending increment is used to disambiguate method calls and object references throughout the semantics.

Let us now consider the corresponding behaviour of the $Rectangle$ class in Fig. 4. In the trace below, $\mathcal{R}$ abbreviates the $Rectangle$ process, $\mathcal{O}$ is the first object generated by the constructor ($\mathcal{O} = \gamma_{\text{RE}}(1)$), and $\mathcal{I}$ is the method instance of the initialiser instantiated with the actual parameters ($\mathcal{I} = (\mathbf{mi}_0 \;\; init_{\rho_{\text{RE}}}(2,1))$).

$$\mathcal{R} \xrightarrow{\text{new(R)?} \quad \text{R}_0.init(2,1)?} \mathcal{R}^+\mathcal{O}^+\mathcal{I} \xrightarrow{\ldots \quad \text{R}_0.\text{return?}} \mathcal{R}'^+\mathcal{O}' \xrightarrow{\text{R}_1.scale(3)? \quad \ldots \quad \text{R}_1.\text{return?}} \mathcal{R}''^+\mathcal{O}''$$

Here the '...' in the trace correspond to the transitions that the object takes in executing first the initialiser and then the body of $scale$. In this instance, these are simply modifications of the fields in $\mathcal{O}$ and $\mathcal{R}$, and in fact all steps become silent. The primed versions of processes $\mathcal{R}$ and $\mathcal{O}$ maintain the same structure but with fields updated according to the execution of the initialiser and $scale$. The trace includes an instance of (4) and two instances of (3).

Having constructed the traces for $Main$ and $\mathcal{R}$ separately, we may now observe the behaviour of their composition, $Main \parallel \mathcal{R}$. This process proceeds by Rule 7, with corresponding invocation and response transitions matching and becoming internal steps. This leaves only the interaction with the environment as the observable transitions in the trace.

$$Main \parallel \mathcal{R} \xrightarrow{\text{input(3)}} (\textbf{state}\ \{r \mapsto \text{R}\} \bullet c) \parallel \mathcal{R}''^{+}\mathcal{O}''$$

3. Sequential code

In this section we describe the semantics of a basic sequential programming language, which includes the usual command types such as assignment, conditional, and iteration. We also include a command type *local state*, which is similar to a variable declaration, except that it also locally maintains the current value of the declared variables. We then extend the language to include exceptions, input and output, and finally top-level programs. The semantics is presented in *small-step* style, meaning that commands, and expression evaluation, may interleave at a fine-grained level.[2]

3.1. Basic definitions

Assume a set of values, Val, and a set of identifiers, $Ident$. We assume the set Val contains the standard boolean and integer values, as well as more complex values as required. The set $Ident$ is a set of identifiers, which are used for variable names, and later for method names.

A $State$ (sometimes called a *store* or *evaluation*) is a partial mapping from variable identifiers to values, $Ident \nrightarrow Val$. We use the notation $\{x \mapsto v, y \mapsto w\}$ to represent the state which maps x to the value v and y to the value w. The set of variables declared by a state σ are those in its domain, written $\text{dom}(\sigma)$, e.g., $\text{dom}(\{x \mapsto v, y \mapsto w\}) = \{x, y\}$.

3.2. Expressions

The abstract syntax of an expression $e \in Expr$ is given in Fig. 6. An expression is either a value, a variable, or the addition of two expressions. We also allow other arithmetic and logical operators, that for space reasons we do not explicitly declare.

Expressions are evaluated in a series of steps from left-to-right, where a 'step' may be looking up the value of a variable in context, or calculating the result of some arithmetic operation. We have chosen a level of atomicity where the retrieval or update of a single variable can be performed atomically, and hence interference is possible during the evaluation of expressions that involve two or more variables. The level of atomicity is controlled by the structure of the labels, and these can be adapted for different atomicity assumptions, for instance, letting labels range over general expressions (or relations) [CH09]. In Sect. 9 we present a locking scheme based on Java's monitors for managing interference that may arise.

The semantics of expression evaluation is given through a labelled transition relation.

$$\longrightarrow : Label \to (Expr \leftrightarrow Expr)$$

It is defined as the least relation that satisfies the operational rules for expressions (including Rules 8 and 9, described below). A transition $(\ell, e_1, e_2) \in \longrightarrow$, written $e_1 \xrightarrow{\ell} e_2$, states that expression e_1 (partially) evaluates to expression e_2, provided that the label ℓ is allowed by the context.

There are three basic forms for a label $\ell \in Label$ as shown in Fig. 6: τ, for an internal step; a test $x = v$, where $x \in Ident$ and $v \in Val$; and an update $x := v$. The label type τ does not depend on nor affect the context. The label type $x = v$ requires that x has the value v in context, while a label $x := v$ requires that the context updates x to the value v.

[2] Mosses has demonstrated [Mos04b] that when using big-step semantics, abrupt termination (through exceptions and return statements) becomes harder to express, and hence we prefer the small-step style to big-step.

$$\textit{Assume} \quad e, e_i, b \in \textit{Expr} \quad v, v_i \in \textit{Val} \quad x \in \textit{Ident} \quad c, c_i \in \textit{Cmd} \quad \ell \in \textit{Label}$$

$$e ::= v \mid x \mid e_1 + e_2 \mid \ldots$$
$$c ::= \mathbf{nil} \mid (x := e) \mid (c_1 \,;\, c_2) \mid (\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) \mid (\mathbf{while}\ b\ \mathbf{do}\ c)$$
$$\ell ::= \tau \mid x = v \mid x := v$$

Rule 8 (Evaluate variable).

$$x \xrightarrow{x=v} v$$

Rule 9 (Evaluate addition).

$$(a)\ \frac{e_1 \xrightarrow{\ell} e_1'}{(e_1 + e_2) \xrightarrow{\ell} (e_1' + e_2)} \qquad (b)\ \frac{e_2 \xrightarrow{\ell} e_2'}{(v + e_2) \xrightarrow{\ell} (v + e_2')} \qquad (c)\ \frac{v_1 + v_2 = v}{(v_1 + v_2) \to v}$$

Rule 10 (Assignment).

$$(a)\ \frac{e \xrightarrow{\ell} e'}{(x := e) \xrightarrow{\ell} (x := e')}$$

$$(b)\quad (x := v) \xrightarrow{x := v} \mathbf{nil}$$

Rule 11 (Sequential composition).

$$(a)\ \frac{c_1 \xrightarrow{\ell} c_1'}{(c_1 \,;\, c_2) \xrightarrow{\ell} (c_1' \,;\, c_2)}$$

$$(b)\quad (\mathbf{nil} \,;\, c_2) \longrightarrow c_2$$

Rule 12 (Conditional).

$$(a)\ \frac{b \xrightarrow{\ell} b'}{(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) \xrightarrow{\ell} (\mathbf{if}\ b'\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2)}$$

$$(b)\quad (\mathbf{if}\ \mathit{true}\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) \longrightarrow c_1 \qquad (c)\quad (\mathbf{if}\ \mathit{false}\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) \longrightarrow c_2$$

Rule 13 (While).

$$(\mathbf{while}\ b\ \mathbf{do}\ c) \longrightarrow (\mathbf{if}\ b\ \mathbf{then}\ (c \,;\, \mathbf{while}\ b\ \mathbf{do}\ c)\ \mathbf{else}\ \mathbf{nil})$$

Fig. 6. Semantics of expression evaluation and sequential commands

A variable x evaluates to a value v provided $x = v$ in context (Rule 8, Fig. 6). This transition defines a relationship between every variable and every value, and hence is highly nondeterministic taken in isolation. For instance, assume that there are only two variables, $\mathit{Ident} = \{x, y\}$, and two values, $\mathit{Val} = \{0, 1\}$. Then the complete set of possible transitions according to Rule 8 is as follows.

$$x \xrightarrow{x=0} 0 \qquad x \xrightarrow{x=1} 1 \qquad y \xrightarrow{y=0} 0 \qquad y \xrightarrow{y=1} 1 \tag{5}$$

The rules for evaluating a binary addition enforce a strict left-to-right evaluation order, where the left-most operand must be fully evaluated to a value before the right-most operand is evaluated (Rule 9(a) and (b)). The behaviour of the left-most operand when being evaluated, ℓ, is the behaviour of the whole expression when being evaluated, and similarly for the right-most operand. When both have been evaluated to values, the sum is calculated (Rule 9(c)). Note the usual distinction between the symbol '+' on the bottom line, which is a syntactic construct, and the symbol '+' above the line, which denotes the semantics of addition. This form of transition rule may be used to specify the evaluation of other expressions; in particular we assume similar rules exist for calculating multiplication, subtraction, exponentiation, and inequalities.

As an example, consider the evaluation of expression $x + y$ assuming that *Ident* and *Val* are defined as above. From Rule 9(a), Rule 8 and (5), there are two possible (partial) evaluations as the first step.

$$x + y \xrightarrow{x=0} 0 + y \qquad x + y \xrightarrow{x=1} 1 + y$$

Assume for now that the label $x = 0$ is allowed by the context (the mechanics of this are explained below). Then continuing the evaluation via Rule 9(b), Rule 8 and (5):

$$0 + y \xrightarrow{y=0} 0 + 0 \qquad 0 + y \xrightarrow{y=1} 0 + 1$$

Finally, assuming the label $y = 1$ is allowed by the context (and hence the label $y = 0$ is disallowed), we have the following final transition from Rule 9(c).

$$0 + 1 \xrightarrow{\tau} 1$$

The τ label on the transition indicates that, as expected, the expression $0 + 1$ evaluates to 1 in any context. No more evaluation is possible, since we have reduced the expression to an element of Val. To be more concise, we now elide the label τ from transitions, that is, $e \longrightarrow e'$ iff $e \xrightarrow{\tau} e'$. We call such transitions *internal*.

Combining the above steps gives the evaluation sequence (6), although the sequence (7) is also valid, depending on the value of x and y in context.

$$x + y \xrightarrow{x=0} 0 + y \xrightarrow{y=1} 0 + 1 \longrightarrow 1 \tag{6}$$

$$x + y \xrightarrow{x=1} 1 + y \xrightarrow{y=1} 1 + 1 \longrightarrow 2 \tag{7}$$

3.3. Commands

The abstract syntax of a command $c \in Cmd$ is described in Fig. 6. The special command **nil** indicates a terminated command. It can partake in no further action. An assignment $x := e$ is the standard assignment command. We assume $x \in Ident$ and e is an expression as defined in the previous section (a richer expression syntax for e, and other possibilities for x including as array indexing, is given in [CH11]). Sequential composition, conditional, and while commands are standard. Note that we allow brackets to enclose commands and expressions where it aids readability, and do not consider them part of the syntax of the language.

The *local state* command ($\mathbf{state}\ \sigma \bullet c$) declares variables in the domain of σ to be 'local' to c. This abstract command type corresponds loosely to the concrete syntax of declaring a new variable and providing it with an initial value, which is its value in σ. However, its value changes as it is updated by c. Local states may be nested, and can be used to model shared variables if c contains concurrency. Constants are local variables that never change value.

The semantics of command execution is defined with respect to a transition relation '$\longrightarrow$'.

$$\longrightarrow : Label \to (Cmd \leftrightarrow Cmd)$$

It is the least relation that satisfies the operational rules for commands. We do not syntactically distinguish transitions on expressions from transitions on commands, but we ensure the correct relation is always clear from the types.

The semantics of the language, excluding local states, appears in Fig. 6. The command **nil** can take no transitions, and therefore there is no corresponding rule. The rules for sequential composition, conditional and iteration are standard. The novel rule is for an assignment $x := e$ (Rule 10); the expression e is first evaluated (eventually) to a value, v, then the command terminates with the update $x := v$ exposed in the label.

As an example, consider the following evaluation and execution of an assignment to x, where the assignment expression is evaluated as in (7).

$$(x := x + y) \xrightarrow{x=1} (x := 1 + y) \xrightarrow{y=1} (x := 1 + 1) \longrightarrow (x := 2) \xrightarrow{x:=2} \mathbf{nil}$$

The rules for local state commands are given in Fig. 7. Rule 14 states that a local state is eliminated when its command terminates. Rule 15 states that if c can transition independently of the context, then so can the command ($\mathbf{state}\ \sigma \bullet c$).

Rule 16(a) applies when the command c requires $x = v$ to be true to transition to c', and x is in the domain of σ. A transition is allowed only when v has the correct value for x in σ—transitions with incorrect values are prevented from occurring (pruned). The behaviour no longer depends on the context and so the promoted label is τ. Rule 16(b) applies when x is not in the domain of σ, and hence the label is promoted for checking at some outer level.

$Assume \quad \sigma \in State$

$c ::= \ldots \mid (\mathbf{state}\ \sigma \bullet c)$

Rule 14 (Eliminate state).

$$(\mathbf{state}\ \sigma \bullet \mathbf{nil}) \longrightarrow \mathbf{nil}$$

Rule 15 (Internal step within state).

$$\frac{c \longrightarrow c'}{(\mathbf{state}\ \sigma \bullet c) \longrightarrow (\mathbf{state}\ \sigma \bullet c')}$$

Rule 16 (Test state).

$$(a)\ \frac{c \xrightarrow{x=v} c' \quad x \in \mathrm{dom}(\sigma) \quad \sigma(x) = v}{(\mathbf{state}\ \sigma \bullet c) \longrightarrow (\mathbf{state}\ \sigma \bullet c')} \qquad (b)\ \frac{c \xrightarrow{x=v} c' \quad x \notin \mathrm{dom}(\sigma)}{(\mathbf{state}\ \sigma \bullet c) \xrightarrow{x=v} (\mathbf{state}\ \sigma \bullet c')}$$

Rule 17 (Update state).

$$(a)\ \frac{c \xrightarrow{x := v} c' \quad x \in \mathrm{dom}(\sigma)}{(\mathbf{state}\ \sigma \bullet c) \longrightarrow (\mathbf{state}\ \sigma[x \mapsto v] \bullet c')} \qquad (b)\ \frac{c \xrightarrow{x := v} c' \quad x \notin \mathrm{dom}(\sigma)}{(\mathbf{state}\ \sigma \bullet c) \xrightarrow{x := v} (\mathbf{state}\ \sigma \bullet c')}$$

Fig. 7. Semantics of local states

For example, the following choice for partial evaluation of the expression $x + y$ is valid by Rule 16(a), which also prevents any other evaluation.

$$(\mathbf{state}\ \{x \mapsto 1\} \bullet x := x + y) \longrightarrow (\mathbf{state}\ \{x \mapsto 1\} \bullet x := 1 + y)$$

This is ultimately how the choice between evaluations such as (6) and (7) are made. Although Rule 8 is highly nondeterministic, the nondeterminism is resolved by the constraint in Rule 16(a). This pattern of context resolving nondeterminism generated by an internal command recurs throughout the paper. Note also that this has become an internal step, since the value of x is found locally and does not depend on any outer context.

Rule 17(a) is the rule where state changes occur. If $x := v$ is the behaviour, and x is local to σ, then σ is updated appropriately. This is now context-independent. We let $\sigma[x \mapsto v]$ represent the state which is equal to σ in every argument except x, which it maps to the value v. For example,

$$(\mathbf{state}\ \{x \mapsto 1\} \bullet x := 2) \longrightarrow (\mathbf{state}\ \{x \mapsto 2\} \bullet \mathbf{nil})$$

In Rule 17(b), c is updating a variable that is not in the local state. There is therefore no change in σ, and the label remains the same, i.e.,

$$(\mathbf{state}\ \{y \mapsto 1\} \bullet x := 2) \xrightarrow{x := 2} (\mathbf{state}\ \{y \mapsto 1\} \bullet \mathbf{nil})$$

3.4. Simplifications for examples

To simplify the presentation of examples, we introduce the definitions and notation given in Fig. 8.

For instance, to reduce the number of steps shown in the execution of a program, we sometimes bundle transitions that involve multiple internal steps. Informally, where ℓs is a sequence of labels, $c \xRightarrow{\ell s} c'$ holds if c can take a series of transitions to evolve to c', with the labels given by ℓs. This is formalised in Definition 8. For instance, from (7),

$$x + y \xrightarrow{x=1 \quad y=1 \quad \tau} 2$$

Definition 8 (Bundled transitions). The relation $c \stackrel{\ell s}{\Longrightarrow} c''$, where ℓs is a finite sequence of labels, is defined as follows:

$$c \stackrel{\langle\rangle}{\Longrightarrow} c \qquad c \stackrel{\ell \quad \ell s}{\Longrightarrow} c'' \Leftrightarrow (c \stackrel{\ell}{\longrightarrow} c' \wedge c' \stackrel{\ell s}{\Longrightarrow} c'')$$

where $\langle\rangle$ denotes the empty sequence, and a list of labels separated by whitespace represents the concatenation of the elements, i.e., '$\ell \quad \ell s$' denotes a non-empty sequence with first element ℓ. When writing traces we tend to omit internal steps (τ), and if a trace is empty or formed entirely from silent steps we leave the label blank.

Definition 9 (Compound assignment).

$$x \mathrel{+}= e \quad \widehat{=} \quad x := x + e$$

Rule 18 (Compound assignment).

$$(x \mathrel{+}= e) \stackrel{x=v}{\longrightarrow} (x := v + e)$$

Rule 19 (Local state promotion).

Command $(\mathbf{state}\ \sigma \bullet c)$ promotes ℓ through c
for ℓ not equal to $x = v, x := v$

Fig. 8. Bundled transitions, compound assignment, and promotion

For brevity when writing sequences of labels, we omit τ labels, that is, if ℓs is a sequence consisting of multiple internal steps and exactly one non-internal step, ℓ, we write $c \stackrel{\ell}{\Longrightarrow} c'$ instead of $c \stackrel{\ell s}{\Longrightarrow} c'$, and if ℓs is composed entirely of τ transitions, we write $c \Rightarrow c'$. For example, we let the left-hand side below abbreviate the right-hand side.

$$(x := v\ ;\ c) \stackrel{x := v}{\Longrightarrow} c \qquad (x := v\ ;\ c) \stackrel{x := v}{\longrightarrow} (\mathbf{nil}\ ;\ c) \longrightarrow c$$

Some of the examples are made shorter by using a *compound assignment* (Definition 9). We may derive Rule 18 from Rule 10.

Finally, we observe that the nested term structure we develop makes use of a large number of *promotion* rules, that is, simple rules that are defined so that the label, ℓ, on a transition of a subcommand, becomes the label on the full command. Rule 9(a), Rule 11(a) and Rule 12(a) are all examples of promotion rules, which apply to any label ℓ. Rule 15 is also a form of promotion—the specific label τ is promoted from a transition of the subcommand c to a transition of the command $(\mathbf{state}\ \sigma \bullet c)$. This is the more general form of promotion law, where a particular language construct, such as $(\mathbf{state}\ \sigma \bullet c)$, has a special relationship with the label types $x = v$ and $x := v$, but otherwise promotes labels such as τ. A more general promotion rule for local states is Rule 19, from which Rule 15 may be derived. It is a short-hand for the following

$$\frac{c \stackrel{\ell}{\longrightarrow} c'}{(\mathbf{state}\ \sigma \bullet c) \stackrel{\ell}{\longrightarrow} (\mathbf{state}\ \sigma \bullet c')} \qquad \text{provided } \ell \neq (x = v) \text{ and } \ell \neq (x := v)$$

Because of the trivial and ubiquitous nature of the promotion rules, we will not mention them in examples.

3.5. Exceptions

We now describe *exceptions*, which are used for handling undefined behaviour at runtime. We will assume that the type *Exception* contains at least the values *divexc*, for division-by-zero errors, and *castexc*, for class casting errors. In Fig. 9 we extend the syntax of expressions, commands, and labels. The '...' in the productions refer to syntax defined earlier, i.e., at the top of Fig. 6.

Assume $exc \in Exception$

$e ::= \ldots \mid \mathbf{throw}\ exc \mid e_1/e_2 \mid \mathbf{input}$
$c ::= \ldots \mid \mathbf{throw}\ exc \mid (\mathbf{try}\ c_1\ \mathbf{catch}\ (exc)\ c_2) \mid \mathbf{output}\ e \mid \mathbf{Program}\ c$
$\ell ::= \ldots \mid \mathsf{throw}(exc) \mid \mathsf{input}\ (v) \mid \mathsf{output}\ (v)$

Rule 20 (Throw).

(a) $\mathbf{throw}\ exc \xrightarrow{\mathsf{throw}(exc)} \mathbf{nil}$ (b) $\mathbf{throw}\ exc \xrightarrow{\mathsf{throw}(exc)} v$

Rule 21 (Division).

(a) Expression (e_1/e_2) promotes ℓ through e_1 (b) Expression (v/e_2) promotes ℓ through e_2

(c) $\dfrac{v_2 \neq 0 \quad v = v_1 \operatorname{div} v_2}{v_1/v_2 \longrightarrow v}$ (d) $\dfrac{v_2 = 0}{v_1/v_2 \xrightarrow{\mathsf{throw}(divexc)} v_1/v_2}$

Rule 22 (Try–catch).

(a) Command $(\mathbf{try}\ c_1\ \mathbf{catch}\ (exc)\ c_2)$ promotes ℓ through c_1 for ℓ not equal to throw

(b) $\dfrac{c_1 \xrightarrow{\mathsf{throw}(exc)} c_1'}{(\mathbf{try}\ c_1\ \mathbf{catch}\ (exc)\ c_2) \longrightarrow c_2}$ (c) $\dfrac{c_1 \xrightarrow{\mathsf{throw}(exc')} c_1' \quad exc' \neq exc}{(\mathbf{try}\ c_1\ \mathbf{catch}\ (exc)\ c_2) \xrightarrow{\mathsf{throw}(exc')} \mathbf{nil}}$

(d) $(\mathbf{try}\ \mathbf{nil}\ \mathbf{catch}\ (exc)\ c_2) \longrightarrow \mathbf{nil}$

Rule 23 (Input from env).

$\mathbf{input} \xrightarrow{\mathsf{input}(v)} v$

Rule 24 (Output to env).

(a) Command $(\mathbf{output}\ e)$ promotes ℓ through e

(b) $\mathbf{output}\ v \xrightarrow{\mathsf{output}(v)} \mathbf{nil}$

Rule 25 (Program).

$$\frac{c \xrightarrow{\ell} c'}{\mathbf{Program}\ c \xrightarrow{\ell} \mathbf{Program}\ c'} \quad \textit{provided}\ \ell \in \{\tau, \mathsf{input}(v), \mathsf{output}(v), \mathsf{throw}(exc)\}$$

Fig. 9. Semantics of exceptions, input and output, and programs

An exception may be explicitly thrown by a **throw** command or expression (Rule 20(a) and (b), respectively).[3] Exceptions may also arise through undefined computation, such as division by 0. This case is given by Rule 21, which states that an expression e_1/e_2 is evaluated first in e_1 then in e_2, and if the denominator is not 0, the final value is the division, otherwise a *divexc* exception is thrown.

A try/catch command $(\mathbf{try}\ c_1\ \mathbf{catch}\ (exc)\ c_2)$ is a kind of context which allows command c_1 to execute normally under most circumstances (Rule 22(a)), unless it throws exception exc, when control is transferred to command c_2 (Rule 22(b)). Rule 22(c) states that exceptions other than exc are promoted, and the try/catch command terminates. Rule 22(d) states that a try/catch command terminates when the main command terminates.

To avoid distraction, throughout the paper we treat exceptions as basic types, deferring until Appendix A a more complete treatment where exceptions are objects, null references are handled, and the *finally* command is defined.

[3] We require a rule for each case as we keep commands and expressions as separate syntactic categories. An alternative would be to treat commands as expressions, as in [KN06].

$c ::= \ldots \mid (c_1 \parallel c_2) \qquad \ell ::= \ldots \mid \ell! \mid \ell?$

Rule 26 (Concurrency).

(a) Command $(c_1 \parallel c_2)$ promotes ℓ through c_1 or c_2

(b) $(\mathbf{nil} \parallel c) \longrightarrow c \qquad (c)$ $(c \parallel \mathbf{nil}) \longrightarrow c$

Rule 27 (Match communication).

$$\frac{c_1 \xrightarrow{\ell!} c_1' \qquad c_2 \xrightarrow{\ell?} c_2'}{(c_1 \parallel c_2) \longrightarrow (c_1' \parallel c_2')}$$

A symmetric rule in c_2 also holds.

Rule 28 (Match self).

$$\frac{c \xrightarrow{\ell!} c' \qquad c' \xrightarrow{\ell?} c''}{c \longrightarrow c''}$$

Fig. 10. Semantics of concurrency

3.6. Input/output and program scope

We extend the language with two commands which abstract input from and output to the environment, and a command for limiting the scope of a program. That is, some basic commands for interacting with an external environment, and delineating a program from the environment. The delineation is important later when we consider communication within an object-oriented system, and need to restrict interactions to just those occurring in the program.

An **input** expression is replaced by some value v, determined by the environment (Rule 23). An **output** e command first evaluates e then terminates with label output (v) (Rule 24). These commands and their label-based semantics is that used by Reynolds for an imperative language [Rey98].

A command (**Program** c) designates the boundary of the program c. It is used as the top-level command enclosing all processes of the object-oriented system. All inter-process communication initiated by processes within c must be with other processes within c, with the exception of the input/output commands defined above. Semantically, the (**Program** c) command acts like the CCS restriction operator, restricting all communication labels from promoting outside of its scope, and hence forcing communication within its scope (the CCS restriction command explicitly specifies the set of events it restricts from the environment, whereas the **Program** command implicitly restricts all communication events). The command also allows internal steps to proceed, e.g., those that are individual steps of processes involving local variables, and it reports uncaught exceptions to the environment. When the scope of a program is clear from context we will tend to omit the **Program** keyword. Other types of communication with the environment, such as remote method/procedure calls, can be defined by making explicit the call types that are not restricted.

4. Communicating processes

We now describe the model for interprocess communication we use throughout the paper to coordinate objects, classes, and method instances. We treat communication abstractly in this section, but later use it to model testing and updating fields, calling methods and constructors, and synchronisation via monitors.

4.1. Concurrency

We introduce the parallel composition operator between two commands, $(c_1 \parallel c_2)$. This operator never appears in sequential code, such as method bodies, but is instead implied by the structure of the program: top-level classes operate in parallel, and objects of those classes also operate in parallel (see Fig. 4). The semantics is given in Rule 26 in Fig. 10: concurrent commands may interleave their steps, and may be eliminated after termination.

For example, consider the following concurrent program in which two commands share a variable g but each have their own local variable h. They compete to set g to the value of their own copy of h.

$$(\mathbf{state}\ \{h \mapsto 0\} \bullet g := h) \quad \| \quad (\mathbf{state}\ \{h \mapsto 1\} \bullet g := h) \tag{10}$$

Taking the first branch, from Rules 17 and 14 we have the following trace.

$$(\mathbf{state}\ \{h \mapsto 0\} \bullet g := h) \xrightarrow{g:=0} \mathbf{nil}$$

The trace contains a single non-internal step $g := 0$. Similarly the second branch generates trace $g := 1$. Local variable h is internal to both processes and hence does not appear in the labels. This aspect of the semantics—using the same variable name in different processes does not cause name clashes—helps to simplify reasoning about local states (fields) of objects and classes.

4.2. Invocation, response, and matching

Labels like τ, $x = v$, and $x := v$, are always interleaved—they are not communication labels. Communication occurs between an *invocation* label, $\ell!$, which *matches*[4] with the corresponding *response* label $\ell?$. That is, excluding the exclamation and question marks, the base label, ℓ, is identical.[5]

This leads to the familiar communication semantics in Rule 27, in which parallel processes combine their actions into a single internal step. In CCS [Mil82], matching labels are given by an event a and its complement $\overline{a}$, rather than the '!' and '?' decorations. In ACP [BK84], matching is determined by an auxiliary function on labels rather than the syntactic matching we do here, and in CSP [Hoa85], multiple processes may synchronise. Since we require only binary synchronisation the CCS-style communication scheme is adopted. As with CCS, unmatched communication labels may also be promoted by parallel composition (Rule 26), but they are restricted from occurring unmatched within a program by Rule 25.

To allow for the case where a process must match with itself, e.g., when an object calls a method of itself, and hence the case where matching does not occur across a parallel composition, we introduce Rule 28. This rule chains together an invocation and the corresponding response of a single process. (Rule 27 may be derived from Rule 28 and Rule 26(a).)

5. Fields

We now take a first step towards object-oriented programming, by introducing *referenceable processes* and fields. These two new command types may be combined to give a basic object containing a local state that may be tested and updated by other processes. The syntax and semantics of fields and references are summarised in Fig. 11.

5.1. Fields

The command type $(\mathbf{fields}\ \sigma \bullet c)$ contains a $State\sigma$ in the context of a command c. The domain of σ are field names. A set of fields reacts to internal operations on its domain in exactly the same manner as a local state command, $(\mathbf{state}\ \sigma \bullet c)$, as stated by Rule 29. It differs from a local state in that other processes that are operating in parallel may query or alter the value of the fields. This is handled semantically through matching on the label types $o.f = v$ and $o.f := v$, respectively, where o is an object reference, described below.

An expression $e.f$ is evaluated by first evaluating the expression e to a reference o, and then the whole expression evaluates to some value v, generating the invocation label $o.f = v!$ (Rule 30). The correct value for v will be determined by the matching responder. A field update command, $e_1.f := e_2$, is executed by first evaluating the expression e_1 to an object reference o and then evaluating e_2 to some value v, before transitioning to $\mathbf{nil}$ with invocation label $o.f := v!$ (Rule 31).

Matching response transitions are given by the fields command, which may respond with the value of the specified field (Rule 32), or update the value of a field (Rule 33). To keep a separation of concerns, the reference of the local object is kept separate from the local fields command. Hence, the response transitions use the placeholder α, which will be replaced by the local reference after the label is promoted (see below).

[4] We use *match* rather than the usual process-algebraic term *synchronise* for this sort of interprocess communication to avoid confusion with the `synchronized` keyword of Java.

[5] Note that although labels with multiple decorations ($\ell!?$) are legal in the syntax in Fig. 10, such labels would be nonsensical and are never generated by the rules.

$$\begin{array}{l} \textit{Assume} \quad f \in \textit{Ident} \quad o, \alpha \in \textit{Ref} \\ e ::= \ldots \mid e.f \\ c ::= \ldots \mid e_1.f := e_2 \mid (\mathbf{fields}\ \sigma \bullet c) \mid (\mathbf{ref}\ o \bullet c) \\ \ell ::= \ldots \mid o.f = v \mid o.f := v \end{array}$$

Rule 29 (Fields as local state).

$$\frac{(\mathbf{state}\ \sigma \bullet c) \xrightarrow{\ell} (\mathbf{state}\ \sigma' \bullet c')}{(\mathbf{fields}\ \sigma \bullet c) \xrightarrow{\ell} (\mathbf{fields}\ \sigma' \bullet c')}$$

Rule 30 (Field test).

(a) Expression $(e.f)$ promotes ℓ through e

(b) $o.f \xrightarrow{o.f=v!} v$

Rule 31 (Field update).

(a) Command $(e_1.f := e_2)$ promotes ℓ through e_1

(b) Command $(o.f := e_2)$ promotes ℓ through e_2

(c) $(o.f := v) \xrightarrow{o.f := v!} \mathbf{nil}$

Rule 32 (Respond to field test).

$$\frac{f \in \mathrm{dom}(\sigma) \quad \sigma(f) = v}{(\mathbf{fields}\ \sigma \bullet c) \xrightarrow{\alpha.f=v?} (\mathbf{fields}\ \sigma \bullet c)}$$

Rule 33 (Respond to field update).

$$\frac{f \in \mathrm{dom}(\sigma)}{(\mathbf{fields}\ \sigma \bullet c) \xrightarrow{\alpha.f := v?} (\mathbf{fields}\ \sigma[f \mapsto v] \bullet c)}$$

Rule 34 (Install reference).

$$\frac{c \xrightarrow{\ell} c'}{(\mathbf{ref}\ o \bullet c) \xrightarrow{\ell[\alpha \setminus o]} (\mathbf{ref}\ o \bullet c')}$$

Fig. 11. Semantics of fields

The command $(\mathbf{ref}\ o \bullet c)$ states that fields (and other information introduced later) in the command c have the reference o. At this stage, the only relevant information in c are fields, and hence we define a simple object that is not executing any code by nesting fields inside a reference, $(\mathbf{ref}\ o \bullet \mathbf{fields}\ \sigma \bullet \mathbf{nil})$. For instance, we may define a rectangle object $\mathcal{O}$ with reference R and field h (height) as follows.

$$\mathcal{O} = (\mathbf{ref}\ \mathrm{R} \bullet \mathbf{fields}\ \{h \mapsto 3\} \bullet \mathbf{nil})$$

A command that doubles the height of $\mathcal{O}$, $\mathrm{R}.h := \mathrm{R}.h * 2$, may take the following transitions by Rules 30 and 31.

$$\mathrm{R}.h := \mathrm{R}.h * 2 \xrightarrow{\mathrm{R}.h=3!} \mathrm{R}.h := 3 * 2 \longrightarrow \mathrm{R}.h := 6 \xrightarrow{\mathrm{R}.h := 6!} \mathbf{nil} \tag{11}$$

Note that in the first step the value 3 is used. However, the invoking process may also transition similarly for any other value. The correct value will be determined by the responder, with the transitions for incorrect values of field $\mathrm{R}.h$ ignored. This mechanism for determining the correct values is familiar from the communication mechanism of CCS [Mil82].

The matching responses of $\mathcal{O}$ follow from Rules 32 and 33, with the correct reference installed by Rule 34.

$$\mathcal{O} \xrightarrow{\mathrm{R}.h=3?} \mathcal{O} \xrightarrow{\mathrm{R}.h := 6?} \mathcal{O}[h := 6] \tag{12}$$

Above we have used the abbreviation $\mathcal{O}[h := 6]$ for updating the field h of $\mathcal{O}$ to the value 6, i.e., for the process $(\mathbf{ref}\ \mathrm{R} \bullet \mathbf{fields}\ \{h \mapsto 6\} \bullet \mathbf{nil})$.

The steps in (12) are constructed by combining the rules for fields with the rule for a reference. For example, the first step is constructed below, with the top line justified by Rule 32 and the bottom line by Rule 34.

$$\frac{(\mathbf{fields}\ \{h \mapsto 3\} \bullet \mathbf{nil}) \xrightarrow{\alpha.h=3?} (\mathbf{fields}\ \{h \mapsto 3\} \bullet \mathbf{nil})}{(\mathbf{ref}\ \mathrm{R} \bullet \mathbf{fields}\ \{h \mapsto 3\} \bullet \mathbf{nil}) \xrightarrow{\mathrm{R}.h=3?} (\mathbf{ref}\ \mathrm{R} \bullet \mathbf{fields}\ \{h \mapsto 3\} \bullet \mathbf{nil})}$$

This pattern of inner constructs generating labels with a placeholder (α), which are then instantiated to the correct value through promotion of the labels to the local reference, is common throughout the paper. We assume α is never used as an actual object reference, and only ever appears in response labels prior to instantiation via Rule 34. Any label that does not include the placeholder is unaffected by Rule 34, and is promoted as-is.

The two processes in parallel match their labels by Rule 27, i.e.,

$$(\mathcal{O} \parallel \mathrm{R}.h := \mathrm{R}.h * 2) \Longrightarrow (\mathcal{O}[h := 6] \parallel \mathbf{nil})$$

In addition to operating in parallel, a process may also request the value of its own field, in other words, the invoker and responder may be one and the same. Usually the explicit reference is omitted and the operations are treated as operations on local variables, as in the object

$$(\mathbf{ref}\ \mathrm{R} \bullet \mathbf{fields}\ \{h \mapsto 3\} \bullet h := h * 2)$$

In executing the code $h := h * 2$ the field name h is treated as a local variable and is modified accordingly by Rule 29 and the rules in Fig. 7. However, it may be the case that, given an expression $e.f$ in a method of object o, at runtime e evaluates to o (in particular, if e is the keyword '**this**', discussed in Sect. 7.1). For example, consider the process $\mathcal{O}$ above, extended to explicitly qualify the field h.

$$(\mathbf{ref}\ \mathrm{R} \bullet \mathbf{fields}\ \{h \mapsto 3\} \bullet \mathrm{R}.h := \mathrm{R}.h * 2)$$

The execution of this process uses the same individual steps as in (11), matching to the steps in (12) via Rule 28, resulting in $\mathcal{O}[h := 6]$ after a sequence of internal steps.

6. Objects, classes, and methods

In this section we extend the language to allow method calls. A method call creates a new instance of a method body which executes in the context of the responding object. We begin by introducing the syntax and informally describing the semantics of method calls in Sect. 6.1, and of method definitions in Sect. 6.2. In Sect. 6.3 we state the general template for objects and classes. In Sect. 6.4 we give the formal semantics for method calls and definitions.

6.1. Overview of syntax and method calls

A method call of the form $e.m(\vec{e})$ represents a call of method m with actual parameters $\vec{e}$ on the process referenced by e. The syntax $\vec{e}$ denotes a list of expressions—the rule for evaluating a list from left-to-right is straightforward. A method call may appear either as a command, e.g., $r.scale(2)$, or as an expression, e.g., $x := r.getHt() * 2$. In the latter case the method acts like a function with possible side effects, eventually evaluating to a value.

The responder to a method call creates a *method instance* (or *stack frame*), which is a piece of code identified by a unique method reference. For instance, after a call $\mathrm{R}.getHt()$, where the body of $getHt$ is defined as $\mathbf{return}\ h$, the object R will contain a method instance ($\mathbf{mi}_i\ \ \mathbf{return}\ h$). The method number i is some number chosen locally by R, and in combination with R will give a unique reference within the program.

When the method instance terminates, or explicitly returns as in $getHt$, a message is sent back to the invoker of the method, who may then continue executing. The invoker receives the method return by the $\mathbf{wait}\ o_i$ command, where o_i is the method reference of the call (the combination of the object reference and the method number).

6.2. Method definitions

Consider the following Java-like definition of the class *Rectangle*, which declares the class-level method `getTA`, and the object-level methods `getHt`, `scale`, and `double`.

```
Class Rectangle
  static int tarea;     int h, w;
  static int getTA() { return tarea }
  int getHt() { return h }
  scale(int n) { tarea += (n*n - 1) * h*w; h := h*n; w := w*n; }
  double() { this.scale(2) }
```

In the abstract syntax, a *method definition* is a mapping from a method identifier to a parameterised command. A parameterised command is a function from a sequence of values (the actual parameters) to a command (the method body). The method body contains a local state for the formal parameters, which are initialised to the values of the actual parameters. We take the usual approach of omitting empty formal and actual parameter lists. A set of method definitions, ρ, is declared to be in the scope of a command c by the command $(\mathbf{methods}\ \rho \bullet c)$. This command may respond to an invocation of any of the methods defined in ρ.

For example, the methods $getTA$, $getHt$, $scale$ and $double$ are collected in the set ρ_{RE} (we will extend the set of methods in later sections).

$$\begin{aligned}\rho_{\text{RE}} \mathrel{\widehat{=}} \{\ & getTA \mapsto \mathbf{return}\ tarea, \\ & getHt \mapsto \mathbf{return}\ h, \\ & scale \mapsto \lambda\, N \bullet \mathbf{state}\ \{n \mapsto N\} \bullet \\ & \qquad tarea \mathrel{+}= (n * n - 1) * h * w\ ;\ h := h * n\ ;\ w := w * n, \\ & double \mapsto \mathbf{this}.scale(2)\quad \}\end{aligned} \tag{13}$$

The identifier $getTA$ is mapped to a command that returns the value of class-level field $tarea$, and the identifier $getHt$ is mapped to a command that returns the value of object-level field h. The identifier $scale$ is mapped to a parameterised command: it takes a value, N, which is the value of the actual parameter to $scale$, and this is set as the initial value of formal parameter n. The rest of the method body is the code for $scale$. The identifier $double$ is mapped to a command that calls $scale$ with parameter 2.

The general form of a method definition corresponding to a method declaration in concrete syntax such as "`m(x) { c }`" is the following.

$$m \mapsto (\lambda\, \vec{v} \bullet \mathbf{state}\ (\vec{x} \mapsto \vec{v}) \bullet c)$$

The notation $(\vec{x} \mapsto \vec{v})$ is the state formed by mapping the ith element of $\vec{x}$ to the ith element of $\vec{v}$, and is defined only if the lengths of $\vec{x}$ and $\vec{v}$ are equal. The vector $\vec{v}$ lists the actual parameter values, determined by expression evaluation at runtime. When there are no parameters, as in $getTA$ and $getHt$, the state is elided.

All parameters are passed "by-value"; the semantics of "by-reference" is achieved by passing an object reference as a (value) parameter, as in Java. For instance, consider a method definition '$m(r) \mathrel{\widehat{=}} r.h := 0\ ;\ldots$'. This method takes as a parameter a reference to an object (formal parameter r), and updates field h of that object. In our language this method definition becomes

$$m \mapsto (\lambda\, O \bullet \mathbf{state}\ \{r \mapsto O\} \bullet r.h := 0\ ;\ldots)$$

where O is a placeholder for the actual parameter. Thus given a call "$o.m(\text{R})$", the following new method instance is created in the workspace of the object referenced by o.

$$\mathbf{state}\ \{r \mapsto \text{R}\} \bullet r.h := 0\ ;\ldots$$

The expression $r.h$ in the body of the method instance is locally evaluated to $\text{R}.h$, and thus generates the label $\text{R}.h := 0!$, causing the passed object to have its height set to 0.

6.3. Objects, classes, and workspaces

Each class and object in the system has its own *workspace*, $(\mathbf{wksp}_i\ c)$. In classes, the command c is typically a parallel composition of objects, and in objects, c is typically a parallel composition of method instances. The number i is used to generate unique references (for new objects or method instances), and is strictly increased each time a new command is added to the workspace.[6]

[6] Other mechanisms may be used to generate unique references, such as drawing from a predetermined set; we use strictly increasing natural numbers as it results in a more concise presentation.

An object always has a reference and a workspace, and typically also contains fields. Hence the following command is a standard template for an object, where c is a parallel composition of method instances.

$$(\mathbf{ref}\ o \bullet \mathbf{fields}\ \sigma \bullet \mathbf{wksp}_i\ c) \tag{14}$$

When new objects are created, their fields include a mapping from the special identifier **this** to their object reference (explained in Sect. 7.1). To save space we introduce the following abbreviation for objects, which leaves the mapping for **this** implicit.

$$(\mathbf{obj}\ o(\sigma)_i \bullet c) \mathrel{\widehat{=}} (\mathbf{ref}\ o \bullet \mathbf{fields}\ \sigma[\mathbf{this} \mapsto o] \bullet \mathbf{wksp}_i\ c) \tag{15}$$

For example, $(\mathbf{obj}\ \textsc{r}(\{h \mapsto 3, w \mapsto 3\})_i \bullet c)$ is an abbreviation for

$$(\mathbf{ref}\ \textsc{r} \bullet \mathbf{fields}\ \{\mathbf{this} \mapsto \textsc{r}, h \mapsto 3, w \mapsto 3\} \bullet \mathbf{wksp}_i\ c) \tag{16}$$

A class always has its own reference (the class name), a workspace, its own class-level fields, and a set of methods. Classes also contain a constructor for generating new objects, but we defer further discussion of this until Sect. 7. The standard format of a class is given below, where c is the parallel composition of objects and subclasses of that class.

$$(\mathbf{ref}\ \textsc{cl} \bullet \mathbf{fields}\ \sigma \bullet \mathbf{methods}\ \rho \bullet \mathbf{cstr}\ \gamma \bullet \mathbf{wksp}_j\ c) \tag{17}$$

As with objects, we introduce the following abbreviation for classes of the form (17).

$$\mathbf{class}\ \textsc{cl}(\sigma, \rho, \gamma)_j \bullet c$$

For example, the initial process corresponding to class *Rectangle* is abbreviated as follows.

$$\begin{array}{l}(\mathbf{class}\ \textsc{rect}(\{tarea \mapsto 0\}, \rho_{\textsc{re}}, \gamma_{\textsc{re}})_1 \bullet \mathbf{nil}) \quad abbreviates \\ \quad (\mathbf{ref}\ \textsc{rect} \bullet \mathbf{fields}\ \{tarea \mapsto 0\} \bullet \mathbf{methods}\ \rho_{\textsc{re}} \bullet \mathbf{cstr}\ \gamma_{\textsc{re}} \bullet \mathbf{wksp}_1\ \mathbf{nil})\end{array} \tag{18}$$

In this case, the class *Rectangle* is made up of a class-level field *tarea*, methods $\rho_{\textsc{re}}$, an object constructor $\gamma_{\textsc{re}}$, and an empty workspace, that is, no objects of class *Rectangle* exist initially. We have chosen to initialise the workspace with the number 1 so that the first object is called $\textsc{rect}_1$, although any number could be chosen.

Note that the methods in $\rho_{\textsc{re}}$ contain both instance methods, such as *getHt* and *scale*, that are called on objects, as well as class-level methods, such as *getTA*, that operate on class-level fields and are called on classes. Invocations of the form $\textsc{rect}.getTA()$ will create a method instance in *Rectangle*'s workspace, alongside the *Rectangle* objects.

In addition to the structure of common processes above, we also use some extended notation to indicate containment.

$$\begin{array}{rcl}\mathcal{M} & = & (\mathbf{mi}_k\ cm) \\ \mathcal{O} & = & (\mathbf{obj}\ o(\sigma)_i \bullet co) \\ \mathcal{O}[\mathcal{M}] & = & (\mathbf{obj}\ o(\sigma)_i \bullet co \parallel \mathcal{M}) \\ \mathcal{O}^+\mathcal{M} & = & (\mathbf{obj}\ o(\sigma)_{i+1} \bullet co \parallel \mathcal{M}) \\ \mathcal{C} & = & (\mathbf{class}\ \textsc{cl}(\sigma_{\textsc{cl}}, \rho, \gamma)_j \bullet cc) \\ \mathcal{C}[\mathcal{O}] & = & (\mathbf{class}\ \textsc{cl}(\sigma_{\textsc{cl}}, \rho, \gamma)_j \bullet cc \parallel \mathcal{O}) \\ \mathcal{C}^+\mathcal{O} & = & (\mathbf{class}\ \textsc{cl}(\sigma_{\textsc{cl}}, \rho, \gamma)_{j+1} \bullet cc \parallel \mathcal{O})\end{array} \tag{19}$$

Here $\mathcal{M}$ is a method instance with reference number k executing command cm. Process $\mathcal{O}$ is an object with command co in its workspace, while $\mathcal{O}[\mathcal{M}]$ is the same object but with method instance $\mathcal{M}$ explicit in its workspace. Object $\mathcal{O}^+\mathcal{M}$ is $\mathcal{O}$ after the addition of $\mathcal{M}$ to its workspace (hence, the count in its workspace is incremented). Class $\mathcal{C}$ takes the standard form, and the notation $\mathcal{C}[\mathcal{O}]$ and $\mathcal{C}^+\mathcal{O}$ for classes and objects are analogous to the corresponding notation for objects and method instances, above. We let a subscripted process, e.g., $\mathcal{C}_i$, indicate a process with all elements of $\mathcal{C}$ subscripted by i.

The structure of a program is essentially the parallel composition of classes. Typically there is also some top-level '*Main*' code to start the program, and there may be global variables. We give such a structure below for a program with n top-level classes and globals in σ.

$$\mathbf{Program}\ (\mathbf{state}\ \sigma \bullet \mathcal{C}_1 \parallel \ldots \parallel \mathcal{C}_n \parallel Main)$$

Classes may have subclasses, and during execution objects and method instances are created. For instance, below objects $\mathcal{O}_1$ and $\mathcal{O}_2$ are of class $\mathcal{C}_1$, and $\mathcal{O}_3$ is object of class $\mathcal{S}C$, which is a subclass of $\mathcal{C}_n$. Object $\mathcal{O}_2$ is currently executing method instance $\mathcal{M}_1$ (and possibly others).

Program (**state** $\sigma \bullet \mathcal{C}_1[\mathcal{O}_1 \parallel \mathcal{O}_2[\mathcal{M}_1]] \parallel \ldots \parallel \mathcal{C}_n[\mathcal{S}C[\mathcal{O}_3]] \parallel \mathit{Main}$)

As described in Sect. 4.1, interactions between processes occur through a CCS-like event mechanism. We can specialise Rule 27 for structures such as that above, indicating the matching of two objects.

$$\frac{\mathcal{O}_1 \xrightarrow{\ell!} \mathcal{O}_1' \qquad \mathcal{O}_2 \xrightarrow{\ell?} \mathcal{O}_2'}{\mathcal{C}_1[\mathcal{O}_1] \parallel \mathcal{C}_2[\mathcal{O}_2] \xrightarrow{\tau} \mathcal{C}_1[\mathcal{O}_1'] \parallel \mathcal{C}_2[\mathcal{O}_2']} \qquad \frac{\mathcal{O}_1 \xrightarrow{\ell!} \mathcal{O}_1' \qquad \mathcal{O}_2 \xrightarrow{\ell?} \mathcal{O}_2'}{\mathcal{C}[\mathcal{O}_1 \parallel \mathcal{O}_2] \xrightarrow{\tau} \mathcal{C}[\mathcal{O}_1' \parallel \mathcal{O}_2']} \tag{20}$$

The derived rule below shows an object $\mathcal{O}_1$ in class $\mathcal{C}_1$ calling method m in object $\mathcal{O}_2$ of class $\mathcal{C}_2$. Object $\mathcal{O}_2$ is extended to include a new method instance $\mathcal{M}$, which the premise of the rule constrains to correspond to the definition of m in class $\mathcal{C}_2$ (contained in ρ_2).

$$\frac{\mathcal{O}_1 \xrightarrow{o_{2i}.m(\vec{v})!} \mathcal{O}_1' \qquad m \in \mathrm{dom}(\rho_2) \qquad \mathcal{M} = (\mathbf{mi}_i \;\; m_{\rho_2}(\vec{v}))}{\mathcal{C}_1[\mathcal{O}_1] \parallel \mathcal{C}_2[\mathcal{O}_2] \xrightarrow{\tau} \mathcal{C}_1[\mathcal{O}_1'] \parallel \mathcal{C}_2[\mathcal{O}_2{}^{+}\mathcal{M}]] } \tag{21}$$

This rule uses the method invocation label $o_{2i}.m(\vec{v})!$ which is similar to a method call on object o_2 in command syntax ($o_2.m(\vec{v})$) with the addition of the method number i to uniquely identify this method call in the system (the number i is decided by the responder $\mathcal{O}_2$). In the rest of this section we provide the underlying rules for constructing more powerful rules such as this.

6.4. Semantics

The syntax and semantics of invoking and responding to a method call, and returning from a method, are given in Fig. 12, and the semantics for method instances are given in Fig. 13.

Rule 35 states that after first evaluating the reference e and the actual parameters $\vec{e}$, a method call is replaced by a **wait** o_i command (or expression), where o_i is the unique reference for the call, and which is blocked until the method returns. The method reference o_i is formed from o, the responding object, and i, a number chosen by the responder. The label for the method call is the invocation $o_i.m(\vec{v})!$, which, apart from the addition of i, is identical to the syntax of the command. Rule 35(d) makes explicit that unqualified method calls are interpreted as calls to self.

Rule 36 states that the **return** and **return** e commands return control, or control and a value, respectively, to the calling process. A **return** command does not know the reference for the method instance in which it is executing, and uses a label with the object or class reference placeholder α, subscripted by a method number i. The placeholder is replaced with the correct reference as the label is promoted by Rule 34, and the correct method number is selected by Rule 41(a), below.

The labels generated by **return** commands are matched by the corresponding **wait** o_i command in the suspended caller (Rule 37(a) and (b)). If an exception is returned instead of a value (see below), the method call is replaced by throwing that exception (Rule 37(c)).

Combining Rules 35 and 37 we have the following characteristic traces of method calls, as commands and as expressions.

$$o.m(\vec{v}) \xrightarrow{o_i.m(\vec{v})! \quad o_i.\mathsf{return}?} \mathbf{nil} \qquad o.m(\vec{v}) \xrightarrow{o_i.m(\vec{v})! \quad o_i.\mathsf{return}(v)?} v \tag{22}$$

First the method is invoked, and then execution is suspended until the return signal is received.

Let us now consider a process responding to a method call. Recall the definition of $\mathcal{C}[\mathcal{O}]$ from (19), which is class $\mathcal{C}$ containing object $\mathcal{O}$ in its workspace. Then we expect to derive the following rule, where the process responds to a call of method $m \in \mathrm{dom}(\rho)$ (recall ρ contains the method definitions of class $\mathcal{C}$) by extending the workspace of $\mathcal{O}$ to include a new method instance.

$$\begin{array}{l} \textit{Assume} \quad i \in \mathbb{N} \quad m \in \textit{Ident} \quad \vec{e} \in \mathrm{seq}\, \textit{Expr} \quad \vec{v} \in \mathrm{seq}\, \textit{Val} \quad d \in \textit{Cmd} \\ \qquad\qquad \rho \in (\textit{Ident} \nrightarrow (\mathrm{seq}\, \textit{Val} \nrightarrow \textit{Cmd})) \qquad (\textit{let}\ m_\rho(\vec{v})\ \textit{abbreviate}\ \rho(m)(\vec{v})) \\ e ::= \ldots \mid m(\vec{e}) \mid e.m(\vec{e}) \mid \mathbf{wait}\ o_i \\ c ::= \ldots \mid m(\vec{e}) \mid e.m(\vec{e}) \mid \mathbf{wait}\ o_i \mid \mathbf{return} \mid \mathbf{return}\, e \\ \qquad\qquad \mid (\mathbf{wksp}_i\ c) \mid (\mathbf{methods}\ \rho \bullet c) \mid (\mathbf{mi}_i\ c) \\ \ell ::= \ldots \mid o_i.m(\vec{v}) \mid o_i.m(\vec{v}) \mathrel{\widehat{=}} d \mid o_i.\mathsf{return} \mid o_i.\mathsf{return}(v) \mid o_i.\mathsf{return}(exc) \end{array}$$

Rule 35 (Method call).

(a) Command $e.m(\vec{e})$ promotes ℓ through e

(b) Command $o.m(\vec{e})$ promotes ℓ through $\vec{e}$

(c) $o.m(\vec{v}) \xrightarrow{o_i.m(\vec{v})!} \mathbf{wait}\ o_i$ $\quad$ (d) $m(\vec{e}) \xrightarrow{\mathbf{this}=o} o.m(\vec{e})$

Rule 36 (Return).

(a) $\mathbf{return} \xrightarrow{\alpha_i.\mathsf{return}!} \mathbf{nil}$

(b) Command $(\mathbf{return}\ e)$ promotes ℓ through e

(c) $\mathbf{return}\ v \xrightarrow{\alpha_i.\mathsf{return}(v)!} \mathbf{nil}$

Rule 37 (Wait).

(a) $\mathbf{wait}\ o_i \xrightarrow{o_i.\mathsf{return}?} \mathbf{nil}$

(b) $\mathbf{wait}\ o_i \xrightarrow{o_i.\mathsf{return}(v)?} v$

(c) $\mathbf{wait}\ o_i \xrightarrow{o_i.\mathsf{return}(exc)?} \mathbf{throw}\ exc$

Rule 38 (Workspace).

(a) Command $(\mathbf{wksp}_i\ c)$ promotes ℓ through c

(b) $(\mathbf{wksp}_i\ c) \xrightarrow{\alpha_i.m(\vec{v}) \mathrel{\widehat{=}} d?} (\mathbf{wksp}_{i+1}\ c \parallel d)$

Rule 39 (Respond to method call).

(a) Command $(\mathbf{methods}\ \rho \bullet c)$ promotes ℓ through c unless $\ell = (o_i.m(\vec{v}) \mathrel{\widehat{=}} d?)$, and $m \in \mathrm{dom}\, \rho$

(b) $$\frac{c \xrightarrow{o_i.m(\vec{v}) \mathrel{\widehat{=}} d?} c' \quad m \in \mathrm{dom}(\rho) \quad d = (\mathbf{mi}_i\ m_\rho(\vec{v}))}{(\mathbf{methods}\ \rho \bullet c) \xrightarrow{o_i.m(\vec{v})?} (\mathbf{methods}\ \rho \bullet c')}$$

Fig. 12. Semantics of method invocation and response

$$\frac{m \in \mathrm{dom}(\rho) \qquad \mathcal{M} = (\mathbf{mi}_i\ m_\rho(\vec{v}))}{\mathcal{C}[\mathcal{O}] \xrightarrow{o_i.m(\vec{v})?} \mathcal{C}[\mathcal{O}^+\mathcal{M}]} \tag{23}$$

This rule can be derived from Rules 38 and 39. Rule 38 states that a workspace may be extended to include a new method instance, using the label $o_i.m(\vec{v}) \mathrel{\widehat{=}} d?$, which does not quite match with the invocation label due to the "$\mathrel{\widehat{=}} d$" part. This part of the label is stripped once the correct value for d is found after being promoted to the level of the method definition. Rule 39(a) states that a method declaration promotes all labels except those generated by a workspace, where the method name m is defined in ρ. This special case is handled by Rule 39(b), which allows only those labels in the form $o_i.m(\vec{v}) \mathrel{\widehat{=}} d?$ where m is defined within ρ and the method body, d, is $(\mathbf{mi}_i\ m_\rho(\vec{v}))$, i.e., represents an instantiation of m with actual parameters $\vec{v}$. The label is promoted but with the '$\mathrel{\widehat{=}} d$' part stripped, and matching may therefore occur. Note that if the object is within nested method declarations that define m, the innermost definition will be dynamically selected: this is the semantics of dynamic dispatch, and is discussed in more detail in Sect. 8.

Rule 40 (Method instance promotes).

Command $(\mathbf{mi}_i \;\; c)$ promotes ℓ through c
for ℓ not equal to return, throw, release

Rule 41 (End method).

$$(a)\;\; (\mathbf{mi}_i \;\; \mathbf{nil}) \xrightarrow{\alpha_i.\mathsf{return}!} \mathbf{nil}$$

$$(b)\;\; \frac{c \xrightarrow{\alpha_i.\mathsf{return}!} c'}{(\mathbf{mi}_i \;\; c) \xrightarrow{\alpha_i.\mathsf{return}!} \mathbf{nil}} \qquad (c)\;\; \frac{c \xrightarrow{\alpha_i.\mathsf{return}(v)!} c'}{(\mathbf{mi}_i \;\; c) \xrightarrow{\alpha_i.\mathsf{return}(v)!} \mathbf{nil}} \qquad (d)\;\; \frac{c \xrightarrow{\mathsf{throw}(exc)} c'}{(\mathbf{mi}_i \;\; c) \xrightarrow{\alpha_i.\mathsf{return}(exc)!} \mathbf{nil}}$$

Fig. 13. Semantics of method instances

The transition in (23) can therefore be constructed as follows, from the interaction of the workspace in $\mathcal{O}$ and the method definitions in $\mathcal{C}$. Let $\mathcal{M}$ abbreviate $(\mathbf{mi}_i \;\; m_\rho(\vec{v}))$.

$$\frac{\dfrac{(\mathbf{wksp}_i \;\; c) \xrightarrow{\alpha_i.m(\vec{v})\widehat{=}\mathcal{M}?} (\mathbf{wksp}_{i+1} \;\; c \parallel \mathcal{M})}{\mathcal{O} \xrightarrow{o_i.m(\vec{v})\widehat{=}\mathcal{M}?} \mathcal{O}^+\mathcal{M}}}{\mathcal{C}[\mathcal{O}] \xrightarrow{o_i.m(\vec{v})?} \mathcal{C}[\mathcal{O}^+\mathcal{M}]}$$

The top line is justified by Rule 38(b); the second line is justified by Rule 34; and the bottom line is justified by Rule 39(b) and $m \in \mathrm{dom}(\rho)$. (Also recall the definitions from (19).)

The semantics of a method instance $(\mathbf{mi}_i \;\; c)$ is given in Fig. 13. It executes until the body terminates, returns a value, throws an exception, or releases control[7] (Rule 40). Rule 41(a) states that when the method body terminates, control is returned to the invoker of the method and the method instance terminates. Rule 41(b) and (c) states that if c returns then the method instance is (abruptly) terminated and the return label is promoted. The nondeterminism in the choice of method number (i) introduced by Rule 36 is resolved here. Uncaught exceptions generated by a method instance are returned to the invoker, and the method instance is (abruptly) terminated (Rule 41(d)).

Consider the following derived rules which more clearly show the execution of a method instance. Assume the trace ℓs does not contain return, throw, or release labels, that is, does not specify an abruptly terminated execution.

$$\frac{c \overset{\ell s}{\Longrightarrow} \mathbf{nil}}{(\mathbf{mi}_i \;\; c) \overset{\ell s \quad \alpha_i.\mathsf{return}!}{\Longrightarrow} \mathbf{nil}} \qquad \frac{c \overset{\ell s \quad \alpha_i.\mathsf{return}!}{\Longrightarrow} c'}{(\mathbf{mi}_i \;\; c) \overset{\ell s \quad \alpha_i.\mathsf{return}!}{\Longrightarrow} \mathbf{nil}} \qquad \frac{c \overset{\ell s \quad \mathsf{throw}(exc)}{\Longrightarrow} c'}{(\mathbf{mi}_i \;\; c) \overset{\ell s \quad \alpha_i.\mathsf{return}(exc)!}{\Longrightarrow} \mathbf{nil}}$$

The first states that when the method body c terminates, the method instance terminates and returns control to the invoker. The other two rules cover abrupt termination, via a return statement or an exception. In both cases, the method instance terminates and control is passed back to the invoker.

7. Constructors

The construction of a new object is a two-step procedure: first a new object is created with an empty workspace, and then the appropriate initialisation code is invoked as a method call. In Sect. 7.1 we describe constructor functions that are used to create new objects, in Sect. 7.2 we describe initialisation code, and in Sect. 7.3 we describe invoking and returning from a constructor.

[7] The release label type is used when starting multiple threads and discussion is deferred until Sect. 9.

7.1. Object constructors

A constructor command ($\mathbf{cstr}\ \gamma \bullet c$) defines the constructor γ in the context of c. Function γ is of type $\mathbb{N} \to Cmd$, where the natural number is used to construct a unique reference for the new object. The general form of a constructor γ for a class CL is as follows:

$$\lambda\, j \bullet (\mathbf{ref}\ \mathrm{CL}_j \bullet \mathbf{fields}\ \sigma_{\mathrm{CL}}[\mathbf{this} \mapsto \mathrm{CL}_j] \bullet \mathbf{wksp}_0\ \mathbf{nil})$$

Hence, the new object's reference is the class name subscripted by the parameter j, its fields have the initial values given by the state σ_{CL}, and includes the field **this**, a special member of *Ident* that is used for storing the reference of the object. The workspace is initially empty. For instance, the constructor for class *Rectangle* is defined as follows (using (15)).

$$\gamma_{\mathrm{RE}} \mathrel{\widehat{=}} \lambda\, j \bullet (\mathbf{obj}\ \mathrm{RECT}_j(\{h \mapsto 0, w \mapsto 0\})_0 \bullet \mathbf{nil}) \tag{24}$$

7.2. Initialisers

Initialisers are special methods that are invoked at object creation time (usually to set up specific values of the object's fields). Consider the following extension of class *Rectangle* with a parameterless, empty constructor, and a second constructor that sets the height and width.

```
Class Rectangle
  ...
  Rectangle() { }
  Rectangle(int ht, int wd) {
     h := ht; w := wd; tarea += h*w }
```

All initialiser methods are named by the reserved identifiers $init_n \in Ident$, where $n \in \mathbb{N}$ is the number of parameters;[8] hence the initialiser that takes no parameters is named $init_0$. All initialisers must be defined to terminate with a **return this** command (although this need not be the case in the concrete code, as above). If there is no code associated with a constructor, as with $init_0$ above, the body of the initialiser is exactly **return this**, i.e., the first initialiser corresponds to the method ($init_0 \mapsto \mathbf{return\ this}$).

The other constructor takes 2 parameters, and hence is represented as a method called $init_2$.

$$\begin{aligned} init_2 \mapsto (\lambda\, H, W \bullet \mathbf{state}\ \{ht \mapsto H, wd \mapsto W\} \bullet \\ h := ht\ ;\ w := wd\ ;\ tarea \mathrel{+}= h * w\ ;\ \mathbf{return\ this}) \end{aligned} \tag{25}$$

In the above, H and W are the actual parameters, ht and wd are the formal parameters, and h and w are the local fields.

The initialisers for a class appear in the method declaration for that class. This allows them to be overridden by subclasses as with standard methods. For instance, the methods of *Rectangle* are extended with the initialisers defined above.

$$\begin{aligned} \rho_{\mathrm{RE}} \mathrel{\widehat{=}} \{\ & init_0 \mapsto \mathbf{return\ this}, \quad init_2 \mapsto \ldots, \\ & getTA \mapsto \ldots, \quad getHt \mapsto \ldots, \quad scale \mapsto \ldots, \quad double \mapsto \ldots \ \} \end{aligned} \tag{26}$$

7.3. Invoking and responding to new commands

The syntax and semantics of creating new objects are given in Fig. 14. The syntax of a constructor invocation is $\mathbf{new}\ \mathrm{CL}(\vec{e})$, where CL is a reference to a class, and $\vec{e}$ lists the actual parameters to the constructor. An invocation $\mathbf{new}\ \mathrm{CL}(\vec{v})$ transitions with a label of $\mathsf{new}(\mathrm{CL})\vec{v}j$, where CL is the class reference and j is a number determined by the responder. Their combination, CL_j, is the reference of the new object. Hence, to complete the construction, the call is replaced by an invocation of the appropriate initialiser, $init_{\#\vec{e}}(\vec{e})$, as determined by the number of parameters.

[8] Because this is an untyped setting, we distinguish different initialisers by the number of parameters only. In a typed setting, initialisers may be distinguished by their full signature.

$$\textit{Assume} \quad \text{CL} \in \textit{Ref} \quad j \in \mathbb{N} \quad \gamma \in \mathbb{N} \to \textit{Cmd}$$

$$e ::= \ldots \mid \mathbf{new}\ \text{CL}(\vec{e}) \qquad c ::= \ldots \mid (\mathbf{cstr}\ \gamma \bullet c) \qquad \ell ::= \ldots \mid \mathsf{new}\ \text{CL}_j \mathrel{\widehat{=}} d \mid \mathsf{new}\ \text{CL}_j$$

Rule 42 (New object).

$$\mathbf{new}\ \text{CL}(\vec{e}) \xrightarrow{\mathsf{new}\ \text{CL}_j!} \text{CL}_j.\mathit{init}_{\#\vec{e}}(\vec{e})$$

Rule 43 (New object in workspace).

$$(\mathbf{wksp}_j\ c) \xrightarrow{\mathsf{new}\ \alpha_j \mathrel{\widehat{=}} d?} (\mathbf{wksp}_{j+1}\ c \parallel d)$$

Rule 44 (Respond to constructor call).

$$(a) \quad \begin{array}{c}\text{Command } (\mathbf{cstr}\ \gamma \bullet c) \text{ promotes } \ell \text{ through } c \\ \text{unless } \ell = (\mathsf{new}\ \alpha_j \mathrel{\widehat{=}} d?)\end{array} \qquad (b) \quad \frac{c \xrightarrow{\mathsf{new}\ \alpha_j \mathrel{\widehat{=}} \gamma(j)?} c'}{(\mathbf{cstr}\ \gamma \bullet c) \xrightarrow{\mathsf{new}\ \alpha_j?} (\mathbf{cstr}\ \gamma \bullet c')}$$

Fig. 14. Semantics of constructors

The rules may be combined with the rules for method calls to give the following standard trace for a constructor invocation of class CL, with exactly one parameter v. Let o abbreviate CL_j, the unique reference of the new object.

$$\mathbf{new}\ \text{CL}(v) \xrightarrow{\mathsf{new}(o)! \quad o_0.\mathit{init}_1(v)! \quad o_0.\mathsf{return}(o)?} o$$

An object with new, unique, reference o is created, then the initialiser method init_1 is invoked on that object (with method reference o_0), and finally when the initialiser completes it returns the object reference and the **new** expression evaluates to o.

On the responding side, Rule 43 states that a new process, d, may be added to the a workspace (of a class) if it is the result of a constructor call. This rule is similar to Rule 38(b) for adding method instances to a workspace. Rule 44(a) states that, similarly to method declarations, all labels except new class responses are promoted by a constructor declaration. Rule 44(b) defines the response to a constructor invocation. As with method calls, only those labels which contain objects resulting from the appropriate application of function γ are permitted. Thus we may construct the following rule that combines the workspace with the constructor declaration and the class reference to add a new object $\mathcal{O}$ of class $\mathcal{C}$, under the assumption that $\gamma(j) = \mathcal{O}$ (recall the abbreviations in (19)).

$$\frac{\dfrac{(\mathbf{wksp}_j\ c) \xrightarrow{\mathsf{new}(\alpha)j \mathrel{\widehat{=}} \mathcal{O}?} (\mathbf{wksp}_{j+1}\ c \parallel \mathcal{O})}{(\mathbf{cstr}\ \gamma \bullet \mathbf{wksp}_j\ c) \xrightarrow{\mathsf{new}(\alpha)j?} (\mathbf{cstr}\ \gamma \bullet \mathbf{wksp}_{j+1}\ c \parallel \mathcal{O})}}{(\mathbf{ref}\ \text{CL} \bullet \mathbf{cstr}\ \gamma \bullet \mathbf{wksp}_j\ c) \xrightarrow{\mathsf{new}(\text{CL})j?} (\mathbf{ref}\ \text{CL} \bullet \mathbf{cstr}\ \gamma \bullet \mathbf{wksp}_{j+1}\ c \parallel \mathcal{O})}$$

The inference is justified, top to bottom, by Rule 43, Rule 44(b), and Rule 34. By promotion rules, the inference may be extended straightforwardly to apply to classes in standard form (17). Hence, recalling the notation from (19), we have the rule

$$\mathcal{C} \xrightarrow{\mathsf{new}(o)?} \mathcal{C}^+\mathcal{O} \tag{27}$$

where $\gamma(j) = \mathcal{O}$ and o abbreviates CL_j. After creation, object $\mathcal{O}$ may respond to field accesses and method calls in the usual way, starting with the initialiser.

8. Subclassing

In this section we describe how subclasses and the semantics of dynamic dispatch are treated (Sect. 8.1), class-specific method calls (method calls that do not follow the semantics of dynamic dispatch) (Sect. 8.2), and determining the class of an object (Sect. 8.3).

8.1. Subclasses

A subclass process has the same format and functions as a class process, except that a subclass appears as a subprocess of (within the workspace of) its superclass. Therefore the subclass, and all instances of the subclass, have access to the context provided by the super class, i.e., its methods and its class-level variables. This nested structure naturally supports the semantics of dynamic dispatch: when a method call of m is responded to by some instance of a class, the context is consulted (the relevant label promotes) backwards through the subclasses until the closest definition of m is found (Rule 39(a)).

As an example, consider a subclass *Square* of *Rectangle*. It extends *Rectangle* with a new constructor which takes just one parameter, and a new method called *getSide*().

```
Class Square Extends Rectangle
  Square(int s) { super(s, s) }
  int getSide() { return h }
```

The corresponding definitions in the abstract syntax follow the standard template for classes, methods and constructors. The class *Rectangle* now contains the *Square* as a process in its workspace (recall the structure in Fig. 1).

$$Rectangle \mathrel{\widehat{=}} \mathbf{class}\ \text{RECT}(\{tarea \mapsto 0\}, \rho_{\text{RE}}, \gamma_{\text{RE}})_1 \bullet Square \tag{28}$$

$$Square \mathrel{\widehat{=}} \mathbf{class}\ \text{SQ}(\emptyset, \rho_{\text{SQ}}, \gamma_{\text{SQ}})_1 \bullet \mathbf{nil} \tag{29}$$

$$\rho_{\text{SQ}} \mathrel{\widehat{=}} \{\ init_1 \mapsto \lambda\, S \bullet \mathbf{state}\ \{s \mapsto S\} \bullet init_2(s, s), \quad getSide \mapsto \mathbf{return}\, h\ \} \tag{30}$$

$$\gamma_{\text{SQ}} \mathrel{\widehat{=}} \lambda\, j \bullet \mathbf{obj}\ \text{SQ}_j(\sigma_{\text{RE}})_0 \bullet \mathbf{nil} \tag{31}$$

Class *Square* has no class-level fields, and the methods of *Square* are mostly straightforward translations of the concrete syntax, although note that the call $super(s, s)$ in the constructor has been translated to a call to the the 2-place initialiser, which is implicitly inherited from *Rectangle*. The constructor γ_{SQ} is standard, where a new *Square* object has the same fields, h and w, and initial values, of *Rectangle* (σ_{RE}).

Let $\mathcal{O}$ be an object in the workspace of *Square*. Such an object is in the scope of the (inner) method declarations in ρ_{SQ} (29), and the (outer) method declarations in ρ_{RE} (28). By Rule 39(b) any calls to $\mathcal{O}$ on methods declared in ρ_{SQ} will use that definition, regardless of whether the superclass also defines those methods. However, by Rule 39(a) and (b), calls on methods not declared in ρ_{SQ} but declared in ρ_{RE} will be promoted to use the definition in ρ_{RE}.

More formally, let $\mathcal{O}$ be an object of class $\mathcal{C}_2$, which is a subclass of $\mathcal{C}_1$, as given by the process $\mathcal{C}_1[\mathcal{C}_2[\mathcal{O}]]$. Recalling the notation from (19), the methods of $\mathcal{C}_1$ ($\mathcal{C}_2$) are defined in ρ_1 (ρ_2). We may derive the following rules from Rule 39, which formally state the circumstances under which the definition of m is taken from the class or superclass.

$$\frac{m \in \text{dom}(\rho_2) \qquad \mathcal{M} = (\mathbf{mi}_i \ \ m_{\rho_2}(\vec{v}))}{\mathcal{C}_1[\mathcal{C}_2[\mathcal{O}]] \xrightarrow{o_i.m(\vec{v})?} \mathcal{C}_1[\mathcal{C}_2[\mathcal{O}^+\mathcal{M}]]} \tag{32}$$

$$\frac{m \in \text{dom}(\rho_1) \setminus \text{dom}(\rho_2) \quad \mathcal{M} = (\mathbf{mi}_i \ \ m_{\rho_1}(\vec{v}))}{\mathcal{C}_1[\mathcal{C}_2[\mathcal{O}]] \xrightarrow{o_i.m(\vec{v})?} \mathcal{C}_1[\mathcal{C}_2[\mathcal{O}^+\mathcal{M}]]} \tag{33}$$

Rule (32) states that the new method instance in $\mathcal{O}$ is determined by the definition in $\mathcal{C}_2$ (ρ_2), since it is declared there (it is similar in form to (23)), while rule (33) states that the new method instance is determined by the definition in $\mathcal{C}_1$, since m it is not (re)declared in $\mathcal{C}_2$.

$$e ::= \ldots \mid e.\text{CL}\colon m(\vec{e}) \qquad c ::= \ldots \mid e.\text{CL}\colon m(\vec{e})$$
$$\ell ::= \ldots \mid o_i.\text{CL}\colon m(\vec{v}) \mathrel{\widehat{=}} d \mid o_i.\text{CL}\colon m(\vec{v})$$

Rule 45 (Class-specific method call).

(a) Command $e.\text{CL}\colon m(\vec{e})$ promotes ℓ through e

(b) Command $o.\text{CL}\colon m(\vec{e})$ promotes ℓ through $\vec{e}$

(c) $o.\text{CL}\colon m(\vec{v}) \xrightarrow{o_i.\text{CL}:m(\vec{v})!} \mathbf{wait}\ o_i$ $\qquad$ (d) $\text{CL}\colon m(\vec{e}) \xrightarrow{\mathbf{this}=o} o.\text{CL}\colon m(\vec{e})$

Rule 46 (Workspace).

$$(\mathbf{wksp}_i\ c) \xrightarrow{\alpha_i.\text{CL}:m(\vec{v})\mathrel{\widehat{=}}d?} (\mathbf{wksp}_{i+1}\ c \parallel d)$$

Rule 47 (Respond to class-specific method call).

$$\frac{c \xrightarrow{o_i.\text{CL}:m(\vec{v})\mathrel{\widehat{=}}d?} c' \quad meths(c) = \rho \quad m \in \text{dom}(\rho) \quad d = (\mathbf{mi}_i\ m_\rho(\vec{v}))}{(\mathbf{ref}\ \text{CL} \bullet c) \xrightarrow{o_i.\text{CL}:m(\vec{v})?} (\mathbf{ref}\ \text{CL} \bullet c')}$$

Definition 34 ($meths$)**.**

$$\begin{array}{rcl} meths(\mathbf{fields}\ \sigma \bullet c) & = & meths(c) \\ meths(\mathbf{methods}\ \rho \bullet c) & = & \rho \cup meths(c) \\ meths(\mathbf{cstr}\ \gamma \bullet c) & = & meths(c) \\ meths(\mathbf{wksp}_j\ c) & = & \varnothing \end{array} \quad \text{hence } meths(\mathbf{class}\ \text{CL}(\sigma,\rho,\gamma)_j \bullet c) = \rho$$

Fig. 15. Semantics of class-specific methods

8.2. Class-specific method calls

The nested structure of subclasses naturally gives the semantics of dynamic dispatch for method inheritance whenever a call to method m always refers to the most recent definition at runtime. However, when a method m is overridden by a subclass, but the definition of m in the superclass is required, some extra mechanisms must be in place. This situation occurs most commonly when a subclass needs to extend m to handle some extra fields—m in the subclass is implemented by calling $super.m()$ before or after executing its own specific code.

As an example, consider the class *Cube*, which extends *Square* to three dimensions. This is a different sort of subclassing; *Square* refers to a subset of *Rectangle*s, while *Cube* is an extension of *Square* in sense of inheriting methods, but otherwise cubes are not squares (or rectangles).

```
Class Cube Extends Square
  static int tvol := 0;
  int d := 0;
  Cube(int s) { super(s); d := s; tvol += s*s*s }
  int getVolume() { return h * h * h }
  scale(n) { tvol += (n*n*n - 1) * getVolume();
             super.scale(n); d := d * n; }
```

Note that *Cube* declares its own 1-place constructor, which calls the 1-place constructor of *Square*. It also uses the superclass definition of method *scale*, which in this case refers to the declaration in *Rectangle*, since *Square* does not define it.

We generalise the `super` syntax of Java to a *class-specific* method call, written $o.\text{CL}\colon m(\vec{e})$, which states that object o must call the version of method m declared in class CL. Concrete syntax like `super` may be translated into the class-specific abstract syntax straightforwardly.

The declarations for *Cube* follow the standard templates, with the `super` calls replaced by class-specific method calls, that is, the target class name is made explicit. Class *Square* in (29) is extended to contain *Cube* in its workspace.

$$\begin{aligned} \mathit{Cube} \mathrel{\widehat{=}} \ & \mathbf{class}\ \text{CUBE}(\{\mathit{tvol} \mapsto 0\}, \rho_{\text{CU}}, \gamma_{\text{CU}})_1 \bullet \mathbf{nil} \qquad (35)\\ \rho_{\text{CU}} \mathrel{\widehat{=}} \ & \{ \quad \mathit{init}_1 \mapsto \lambda\, S \bullet \mathbf{state}\ \{s \mapsto S\} \bullet \\ & \qquad\qquad \text{SQ:}\ \mathit{init}_1(s)\ ;\ d := s\ ;\ \mathit{tvol} \mathrel{+}= s * s * s \\ & \quad \mathit{getVolume} \mapsto \mathbf{return}\ h * h * h \\ & \qquad \mathit{scale} \mapsto \lambda\, N \bullet \mathbf{state}\ \{n \mapsto N\} \bullet \\ & \qquad\qquad \mathit{tvol} \mathrel{+}= (n * n * n - 1) * \mathit{getVolume}()\ ; \\ & \qquad\qquad \text{RECT:}\ \mathit{scale}(n)\ ;\ d := d * n \quad \} \\ \gamma_{\text{CU}} \mathrel{\widehat{=}} \ & \lambda\, j \bullet \mathbf{obj}\ \text{CUBE}_j(\sigma_{\text{RE}}[d \mapsto 0])_0 \bullet \mathbf{nil} \qquad (36) \end{aligned}$$

To define the semantics of class-specific methods we introduce the syntax as a command and expression, and, following standard method calls, introduce two new label types $o_i.\text{CL}\colon m(\vec{v}) \mathrel{\widehat{=}} d$ and $o_i.\text{CL}\colon m(\vec{v})$ (see Fig. 15).

Rule 47 is syntactically identical to Rule 35 where the method name m is replaced by $\text{CL}\colon m$. Rule 46 is similarly related to Rule 38.

Rule 47 corresponds to Rule 39(b), however, the method call resolution is performed at the reference level rather than the method declaration level. This is so that the class name can be directly linked to its own specific declaration of method m. The methods declared by a class process c are given by the function $\mathit{meths}(c)$ in definition 34, which straightforwardly extracts any method declarations in c, excluding those defined in its subclasses (by ignoring any declarations occurring in its workspace).

For instance, we may derive the following rule from Rules 46 and 47, which is similar to (33) except that it applies whether or not $m \in \text{dom}(\rho_2)$.

$$\frac{m \in \text{dom}(\rho_1) \quad \mathcal{M} = (\mathbf{mi}_i\ \ m_{\rho_1}(\vec{v}))}{\mathcal{C}_1[\mathcal{C}_2[\mathcal{O}]] \xrightarrow{o_i.\text{CL}_1:m(\vec{v})?} \mathcal{C}_1[\mathcal{C}_2[\mathcal{O}^+\mathcal{M}]]}$$

The premise of Rule 47 holds because $\mathit{meths}(\mathcal{C}_1[\ldots]) = \rho_1$ by Definition 34.

Class-specific fields. In the above we have considered class-specific methods, and it seems logical to extend this concept to class-specific fields. That is, the semantics when (subclass) $\mathcal{C}_2$ declares a field f that was declared in its superclass, $\mathcal{C}_1$. Perhaps the most straightforward option is to disallow this possibility entirely, and hence any objects of class $\mathcal{C}_2$ have only one field with base name f. Another option is to allow the redeclaration of f in $\mathcal{C}_2$ only if the type of the redeclared f is a subclass of the type of f in $\mathcal{C}_1$. In this case also, objects of class $\mathcal{C}_2$ have only one field with base name f (and hence in both of these two options, class-specific field references are not necessary).

The approach adopted by Java is to allow redeclarations of fields, with the previously declared versions accessible within a subclass by qualifying with the `super` keyword. In this scheme, the most straightforward approach is to treat each field declaration 'f' in class $\mathcal{C}_1$ in the concrete syntax as a field '$\mathcal{C}_1\colon f$' in the abstract syntax. Then the redeclaration of f in $\mathcal{C}_2$ means that objects of class $\mathcal{C}_2$ have fields named '$\mathcal{C}_1\colon f$' and '$\mathcal{C}_2\colon f$'.

This leaves the question of how to interpret expressions $o.f$ in the concrete syntax, in the case where o is statically declared of type $\mathcal{C}_1$, but may be an instance of $\mathcal{C}_2$ at run time. In the spirit of dynamic dispatch for method calls, one could require $o.f$ to refer to the innermost declaration of f in the class hierarchy of o at runtime. However, presumably for issues of definedness and efficiency, in Java the choice of field is made based on the compile-time type of o. Hence, to follow Java's approach, a field reference $o.f$ would be translated into abstract syntax as $o.\mathcal{C}_1\colon f$, where $\mathcal{C}_1$ is the declared type of o. (Thus, the correct field is chosen at compile-time, not run-time.)

Each of these three approaches can be dealt with in the concrete syntax and hence do not need special treatment in the semantics. Other semantics are possible but outside the scope of this paper.

8.3. Class-based commands

Figure 16 introduces the expression type (e_1 **instOf** e_2) for testing the class of an object. Rule 48 states that first expression e_1 is evaluated to an object reference and then the expression e_2 to a class reference. If the label $o.\mathsf{instOf}(\text{CL})!$ matches with the responder, the entire expression evaluates to *true*, otherwise the label $\neg o.\mathsf{instOf}(\text{CL})!$ must match with the responder, and the entire expression evaluates to *false*.

$e ::= \ldots \mid (e_1 \ \mathbf{instOf} \ e_2) \mid (e_1)\ e_2 \qquad \ell ::= \ldots \mid o.\mathsf{instOf}(\mathrm{CL}) \mid \neg o.\mathsf{instOf}(\mathrm{CL})$

Rule 48 (Instance of).

(*a*) Expression $(e_1 \ \mathbf{instOf} \ e_2)$ promotes ℓ through e_1

(*b*) Expression $(o \ \mathbf{instOf} \ e_2)$ promotes ℓ through e_2

(*c*) $(o \ \mathbf{instOf} \ \mathrm{CL}) \xrightarrow{o.\mathsf{instOf}(\mathrm{CL})!} \mathit{true}$ (*d*) $(o \ \mathbf{instOf} \ \mathrm{CL}) \xrightarrow{\neg o.\mathsf{instOf}(\mathrm{CL})!} \mathit{false}$

Rule 49 (Respond to instance of).

$$(a)\ \frac{o \in \mathit{refs}(c)}{(\mathbf{ref}\ \mathrm{CL} \bullet c) \xrightarrow{o.\mathsf{instOf}(\mathrm{CL})?} (\mathbf{ref}\ \mathrm{CL} \bullet c)} \qquad (b)\ \frac{o \notin \mathit{refs}(c)}{(\mathbf{ref}\ \mathrm{CL} \bullet c) \xrightarrow{\neg o.\mathsf{instOf}(\mathrm{CL})?} (\mathbf{ref}\ \mathrm{CL} \bullet c)}$$

Definition 37 (*refs*).

$$\begin{array}{rcl} \mathit{refs}(\mathbf{nil}) & = & \varnothing \\ \mathit{refs}(x := e) & = & \varnothing \\ & \ldots & \end{array} \qquad \begin{array}{rcl} \mathit{refs}(c_1 \parallel c_2) & = & \mathit{refs}(c_1) \cup \mathit{refs}(c_2) \\ \mathit{refs}(\mathbf{ref}\ o \bullet c) & = & \{o\} \cup \mathit{refs}(c) \\ & \ldots & \end{array}$$

Fig. 16. Semantics of 'instance of' relations

We may use this expression type to define class casting as follows:

$(e_2)\ e_1 \mathrel{\widehat{=}} \mathbf{if}\ (e_1 \ \mathbf{instOf} \ e_2)\ \mathbf{then}\ e_1\ \mathbf{else\ throw}\ \mathit{castexc}$

Rule 49 gives the semantics for responding to an $o.\mathsf{instOf}(\mathrm{CL})$ label. The responding process is the class referenced by CL. If it contains a reference to o, it may transition with a label of $o.\mathsf{instOf}(\mathrm{CL})?$, otherwise it may transition with a label of $\neg o.\mathsf{instOf}(\mathrm{CL})?$.

The rule makes use of the auxiliary function *refs* (Definition 37), that collects all the class and object references contained in c. Its definition is straightforward by structural induction on the language, and hence we give only a representative set of cases. It is similar to extracting the *alphabet* of a process in CSP [Hoa85].

9. Multithreading

We now extend the semantic framework to allow concurrency. Thus far we have presented classes and objects composed in parallel, but because each invoker of a method call is blocked until the responder terminates and returns control, there is no scope for interleaving of actions without explicitly supporting it in the concrete syntax.

The simplest way of introducing concurrency is to allow the parallel operator in the concrete syntax, and thus the *Main* code may be defined to start n parallel threads, or methods may fork into parallel processes. However, such unrestricted concurrency becomes difficult to manage and understand, as processes compete for access to shared data. Furthermore, the correct semantics of abrupt termination via return statements or exceptions is unclear in the general case. For instance, consider the program $(\mathbf{throw}\ e) \parallel c$. The left-hand process abruptly terminates the current method call, which should terminate the right-hand process, c, but c itself may contain nested concurrency and calls to active method instances executing in other objects. Rather than make arbitrary decisions about how to handle such cases, which do not arise in practice since the parallel operator '$\parallel$' is not part of concrete syntax, and to allow greater control over access to shared data, we will be guided by Java's approach to introducing concurrency via *threads*, where new threads are created only via calls to a special *start* method. This method begins a new thread in the responding object, while the invoker may continue executing.

In Sect. 9.1 we describe how a *start* method acquires a new thread reference and allows the invoker to continue executing in parallel. In Sect. 9.2 we describe the behaviour of method calls in the presence of threads. In Sect. 9.3 we describe monitors, which provide synchronised access to processes through the *synchronised block* command type, which is described in Sect. 9.4. Java also allows other operations on threads, such as suspension and interruption, but a complete semantics for Java threads is outside the scope of this paper.

$Assume \quad \mathrm{T}, \phi \in Ref$

$c ::= \ldots \mid (\mathbf{thr}_{\mathrm{T}}\ c) \mid (\mathbf{newtref}\ c) \mid (\mathbf{trefs}_i\ c) \mid \mathbf{release}$
$\ell ::= \ldots \mid o_i^{\mathrm{T}}.m(\vec{v}) \mid \mathsf{newtref} = \mathrm{T} \mid \mathsf{release}$

Rule 50 (Release).

$$(a)\quad \mathbf{release} \xrightarrow{\mathsf{release}} \mathbf{nil} \qquad (b)\ \frac{c \xrightarrow{\mathsf{release}} c'}{(\mathbf{mi}_i\ c) \xrightarrow{\alpha_i.\mathsf{return!}} c'}$$

Rule 51 (New thread reference).

$$(\mathbf{newtref}\ c) \xrightarrow{\mathsf{newtref}=\mathrm{T}} (\mathbf{thr}_{\mathrm{T}}\ c)$$

Rule 52 (Allocate thread reference).

(a) Command $(\mathbf{trefs}_i\ c)$ promotes ℓ through c

$$(b)\ \frac{c \xrightarrow{\mathsf{newtref}=i} c'}{(\mathbf{trefs}_i\ c) \longrightarrow (\mathbf{trefs}_{i+1}\ c')}$$

Rule 53 (Thread reference).

$$\frac{c \xrightarrow{\ell} c'}{(\mathbf{thr}_{\mathrm{T}}\ c) \xrightarrow{\ell[\phi\backslash\mathrm{T}]} (\mathbf{thr}_{\mathrm{T}}\ c')}$$

Rule 54 (End thread instance).

$$(\mathbf{thr}_{\mathrm{T}}\ \mathbf{nil}) \longrightarrow \mathbf{nil}$$

Rule 55 (Threaded call).

$$o.m(\vec{v}) \xrightarrow{o_i^{\phi}.m(\vec{v})!} \mathbf{wait}\ o_i$$

Rule 56 (Respond to threaded method call).

$$\frac{c \xrightarrow{o_i.m(\vec{v}) \widehat{=} d?} c' \quad m \in \mathrm{dom}(\rho) \quad d = (\mathbf{mi}_i\ \mathbf{thr}_{\mathrm{T}}\ m_\rho(\vec{v}))}{(\mathbf{methods}\ \rho \bullet c) \xrightarrow{o_i^{\mathrm{T}}.m(\vec{v})?} (\mathbf{methods}\ \rho \bullet c')}$$

Fig. 17. Semantics of threads

9.1. Java's thread mechanism

To start a new thread in Java, the special *start* method is invoked on subclasses of the built-in `thread` class. The invoker may then continue executing while the *start* method begins its own thread. Since the *start* method is associated with an object, it may be invoked exactly once per (thread) object, or an exception is thrown. We translate a Java *start* method with body c as follows (to avoid distraction we assume *start* is parameterless).

$$\mathbf{if}\ \mathit{started}\ \mathbf{then\ throw}\ \mathit{ThreadExc}\ \mathbf{else}\ (\mathit{started} := \mathit{true}\ ;\ \mathbf{release}\ ;\ \mathbf{newtref}\ \ c)$$

Multiple copies of the body of *start* are prevented by testing an object-level boolean field *started*.[9] Otherwise, *started* is set to *true*, the invoker is *released* to continue execution, and a new thread reference is obtained before executing c. The command **release** (see Rule 50 in Fig. 17) behaves similarly to a **return** command, except the method instance that executes the **release** may continue executing afterwards.[10] Since the method is now independent of the invoker the method number may be discarded. The $(\mathbf{newtref}\ \ c)$ command obtains a new thread reference from a global thread reference allocator, and then executes c in the new thread (Rule 51). The allocator is given by command $(\mathbf{trefs}_i\ \ c)$, where the number i is used for the new thread reference, and is incremented as threads are allocated (Rule 52). Within a program, the allocator is placed at the outermost level, i.e., for a threaded program c, we write $(\mathbf{Program}\ \mathbf{trefs}_i\ \ c)$.

[9] To avoid race conditions on variable *started* the code may be enclosed in a *synchronised* block (see Sect. 9.4).
[10] Jones [Jon07] uses a similar command to introduce concurrency in an object-oriented programming language.

$$c ::= \ldots \mid (\mathbf{mtr}\ (\mathrm{T}, n) \bullet c) \mid (\mathbf{sync}\ e \bullet c) \mid (\lock o \bullet c)$$
$$\ell ::= \ldots \mid o.\mathsf{lock}(\mathrm{T}) \mid o.\mathsf{unlock}(\mathrm{T})$$

Rule 57 (Monitor).

(a) $(\mathbf{mtr}\ (\mathrm{T}, 0) \bullet c) \xrightarrow{\alpha.\mathsf{lock}(\mathrm{T}')?} (\mathbf{mtr}\ (\mathrm{T}', 1) \bullet c)$

(b) $(\mathbf{mtr}\ (\mathrm{T}, n) \bullet c) \xrightarrow{\alpha.\mathsf{lock}(\mathrm{T})?} (\mathbf{mtr}\ (\mathrm{T}, n+1) \bullet c)$

(c) $(\mathbf{mtr}\ (\mathrm{T}, n) \bullet c) \xrightarrow{\alpha.\mathsf{unlock}(\mathrm{T})?} (\mathbf{mtr}\ (\mathrm{T}, n-1) \bullet c)$

(d) Command $(\mathbf{mtr}\ (\mathrm{T}, n) \bullet c)$ promotes ℓ through c

Rule 58 (Synchronised block).

(a) Command $(\mathbf{sync}\ e \bullet c)$ promotes ℓ through e $\quad$ (b) $(\mathbf{sync}\ o \bullet c) \xrightarrow{o.\mathsf{lock}(\phi)!} (\lock o \bullet c)$

(c) Command $(\lock o \bullet c)$ promotes ℓ through c for ℓ not equal to return, throw $\quad$ (d) $(\lock o \bullet \mathbf{nil}) \xrightarrow{o.\mathsf{unlock}(\phi)!} \mathbf{nil}$

Rule 59 (Abrupt termination of synchronised block).

$$(a)\ \frac{c \xrightarrow{\alpha_i.\mathsf{return}(v)} c'}{(\lock o \bullet c) \xrightarrow{o.\mathsf{unlock}(\phi)!} \mathbf{return}\ v} \qquad (b)\ \frac{c \xrightarrow{\mathsf{throw}(exc)} c'}{(\lock o \bullet c) \xrightarrow{o.\mathsf{unlock}(\phi)!} \mathbf{throw}\ exc}$$

Fig. 18. Semantics of monitors and synchronised blocks

9.2. Threads

In a threaded context, method instances contain a *thread reference* in addition to the method reference. A command c (a method body) executing 'within' thread T is written $(\mathbf{thr}_{\mathrm{T}}\ \ c)$. Any method or constructor calls invoked by c will pass the thread reference T to the responder. Furthermore, any attempt by c to obtain a lock on a process will use T as the reference. A command $(\mathbf{thr}_{\mathrm{T}}\ \ c)$ is created when a new thread is started with reference T, or when a method is invoked by a process already within thread T.

Rule 53 states that a command $(\mathbf{thr}_{\mathrm{T}}\ \ c)$ replaces the placeholder thread reference ϕ with the local thread reference (cf. Rule 34). When the body terminates, the thread reference may be eliminated (Rule 54).

In a threaded program, the invoker of a method call or new object passes its thread reference to the responder, and hence the invocation Rule 35 is replaced by its threaded counterpart, Rule 55, which is identical apart from the addition of the placeholder thread reference ϕ to the label. The placeholder is locally instantiated by Rule 53. Similarly, the method response Rule 39 is replaced by its threaded counterpart Rule 56, which installs the thread reference of the invoker as the thread reference for the new method body. The rules for class-specific method calls (Fig. 15) may be extended to handle threads in a similar manner.

The trace of invokers and responders in a threaded context are the same except that the thread reference appears in the method call labels. For instance, the derived rule corresponding to (23) for responding to a method call at the class level is as follows.

$$\frac{m \in \mathrm{dom}(\rho) \quad \mathcal{M} = (\mathbf{mi}_i\ \ \mathbf{thr}_{\mathrm{T}}\ \ m_\rho(\vec{v}))}{\mathcal{C}[\mathcal{O}] \xrightarrow{o_i^{\mathrm{T}}.m(\vec{v})?} \mathcal{C}[\mathcal{O}^+\mathcal{M}]} \tag{38}$$

An alternative approach for passing thread references from invoker to responder would be to have the thread reference as an implicit parameter to every method declaration, and for every method call to implicitly include the local thread reference as an actual parameter. This would eliminate the need to redefine the operational rules for method calls, but at the expense of complicating the abstract syntax.

9.3. Monitors

Access to a process may be managed through a *monitor*, upon which threads compete to set locks and have exclusive access to the process. We introduce a new abstract command $(\mathbf{mtr}\ (\mathrm{T}, n) \bullet c)$, which indicates that c is a lockable process. The number n counts the number of locks thread T currently has on c—the count may be non-zero because the same process may be locked several times in the same thread. The syntax and semantics appear in Fig. 18. The semantics implicitly uses object references to uniquely identify the monitored command c, since this is natural in a object-oriented paradigm, and processes have straightforward access to that reference type.

An object in a threaded context is extended to include a monitor command, that is, we extend the standard form of an object in (14) to:

$$(\mathbf{ref}\ o \bullet \mathbf{fields}\ \sigma \bullet \mathbf{mtr}\ (\mathrm{T}, n) \bullet \mathbf{wksp}_i\ c) \tag{39}$$

Rule 57 states that a new thread can acquire a lock if no other thread has any locks on c, that is, when n is zero, or may acquire a further lock (become *re-entrant*) on c if it already holds the lock. A thread removes a lock by decrementing n. The labels use the object placeholder α, which is instantiated to the enclosing reference by Rule 34.

Let $\mathcal{O}^{\text{🔓}}$ abbreviate an object of form (39) where the lock count is 0, and let $\mathcal{O}^{\text{🔒}}_{\mathrm{T},n}$ abbreviate an object which is locked n times by thread T. Then we may derive the following rules:

$$\mathcal{O}^{\text{🔓}} \xrightarrow{o.\mathsf{lock}(\mathrm{T})?} \mathcal{O}^{\text{🔒}}_{\mathrm{T},1} \qquad \mathcal{O}^{\text{🔒}}_{\mathrm{T},n} \xrightarrow{o.\mathsf{lock}(\mathrm{T})?} \mathcal{O}^{\text{🔒}}_{\mathrm{T},n+1}$$
$$\mathcal{O}^{\text{🔒}}_{\mathrm{T},1} \xrightarrow{o.\mathsf{unlock}(\mathrm{T})?} \mathcal{O}^{\text{🔓}} \qquad \mathcal{O}^{\text{🔒}}_{\mathrm{T},n+1} \xrightarrow{o.\mathsf{unlock}(\mathrm{T})?} \mathcal{O}^{\text{🔒}}_{\mathrm{T},n} \tag{40}$$

9.4. Synchronise blocks

Locks are acquired on monitors through the command $(\mathbf{sync}\ e \bullet c)$, which blocks until it acquires a lock on the reference e, and then executes c (Rule 58). When c terminates, the lock is released. The command $(\text{🔒}\, o \bullet c)$ distinguishes between a synchronised block which has acquired the lock from one which is waiting to acquire the lock.[11]

Putting these rules together we may derive the following rule, where we assume the the trace ℓs does not abruptly terminate (i.e., does not contain a return or throw label).

$$\frac{c \overset{\ell s}{\Longrightarrow} \mathbf{nil}}{(\mathbf{sync}\ o \bullet c) \overset{o.\mathsf{lock}(\phi)!\quad \ell s \quad o.\mathsf{unlock}(\phi)!}{\Longrightarrow} \mathbf{nil}} \tag{41}$$

That is, executing c inside a synchronised block bookends the execution of c with a lock and unlock of the relevant monitor.

If a synchronise block terminates abruptly, i.e., through a return statement or an uncaught exception, it must release its locks before terminating, or the system may deadlock (Rule 59). In the case of an exception inside a synchronised block, we may derive the following transition, where as above we assume ℓs does not abruptly terminate.

$$\frac{c \overset{\ell s \quad \mathsf{throw}(exc)}{\Longrightarrow} c'}{(\mathbf{sync}\ o \bullet c) \overset{o.\mathsf{lock}(\phi)!\quad \ell s \quad o.\mathsf{unlock}(\phi)!\quad \mathsf{throw}(exc)}{\Longrightarrow} \mathbf{nil}} \tag{42}$$

This states that an exception occurring inside a synchronised block causes the block to release its lock, then throw the exception and terminate.

[11] An alternative to explicit locking by the invoker is to build the synchronisation block into the method definition, for instance, *scale* could be defined to contain a synchronised block locking its own monitor.

$$\mathit{scale} \mapsto (\lambda\, N \bullet \mathbf{state}\ \{n \mapsto N\} \bullet (\mathbf{sync}\ \mathbf{this} \bullet \mathit{tarea} \mathrel{+}= \ldots))$$

10. Tool support and analysis

10.1. Simulation

The operational rules provide an abstract mechanism for formally describing fundamental concepts of object-oriented programming. One advantage of formalising a language operationally is that it lends itself to simulation, by executing a program step-by-step through encodings of the rules. The Maude system [CDE+02, MR11] is a powerful tool for rewriting logic computations, and has been used for defining operational semantics for process algebras and imperative and object-oriented languages [Mes00, VMO04, FCMR04, VMO06, CB07].

The translation of the majority of the rules into the Maude syntax is straightforward following the processes given by Verdejo and Mart-Oliet [VMO06]. For example, Rules 9 and 10 may be translated as follows.

```
crl [rule3-a] : (X := E) =>    {L}    X := E'   if E => {L} E' .
 rl [rule3-b] : (X := V) => {X :== V} nil .
```

The keywords (c)`rl` indicate (conditional) rewrite rules, with an optional tag in square brackets to name the rule. The rules themselves are essentially direct translations, where a transition $e \xrightarrow{\ell} e'$ is written `E => {L} E'`, and a rule $\frac{c}{d}$ is written `d if c`.

Many of the rules are highly nondeterministic, and for tractability must be transformed to be deterministic. For instance, consider Rule 8, which generates a transition for every possible value. This style is attractive in the abstract rules since the outer context prunes away redundant transitions, but in an executable environment this is prohibitively expensive. We therefore rewrite the rule to be deterministic, using a placeholder within the program to stand for the value of the variable, and which interacts with the local state deterministically using term replacement. We repeat Rules 8 and 16(a) in (43) below, followed by their transformation into a deterministic version in (44).

$$x \xrightarrow{x=v} v \qquad \frac{c \xrightarrow{x=v} c' \quad x \in \mathrm{dom}(\sigma) \quad \sigma(x) = v}{(\mathbf{state}\ \sigma \bullet c) \longrightarrow (\mathbf{state}\ \sigma \bullet c')} \tag{43}$$

$$x \xrightarrow{x=\kappa} \kappa \qquad \frac{c \xrightarrow{x=\kappa} c' \quad x \in \mathrm{dom}(\sigma) \quad \sigma(x) = v}{(\mathbf{state}\ \sigma \bullet c) \longrightarrow (\mathbf{state}\ \sigma \bullet c'[\kappa \backslash v])} \tag{44}$$

In the above κ is a reserved constant, appearing only in these rules, used as a placeholder for (the value of) x; the local state determines the value v for x, and hence v is textually substituted into c' in place of κ (given by the notation $c'[\kappa \backslash v]$). This makes the rule deterministic, i.e., for a variable x there is exactly one transition possible (note that κ is redundant in the labels, but we keep it for comparison purposes). Following the same approach the more complex rules may also be transformed to eliminate nondeterminism in the labels, for instance, Rules 38 and 39 (repeated below in (45)), defining the interaction of the workspace and method definition when responding to a method call, may be transformed as shown in (46), where the reserved constant η is used as a placeholder for the new method body.

$$(\mathbf{wksp}_i\ c) \xrightarrow{\alpha_i.m(v)\widehat{=}d?} \mathbf{wksp}_{i+1}\ c \parallel d \qquad \frac{c \xrightarrow{o_i.m(\vec{v})\widehat{=}d?} c' \quad m \in \mathrm{dom}(\rho) \quad d = (\mathbf{mi}_i\ m_\rho(\vec{v}))}{(\mathbf{methods}\ \rho \bullet c) \xrightarrow{o_i.m(\vec{v})?} (\mathbf{methods}\ \rho \bullet c')} \tag{45}$$

$$(\mathbf{wksp}_i\ c) \xrightarrow{\alpha_i.m(v)\widehat{=}\eta?} \mathbf{wksp}_{i+1}\ c \parallel \eta \qquad \frac{c \xrightarrow{o_i.m(\vec{v})\widehat{=}\eta?} c' \quad m \in \mathrm{dom}(\rho) \quad d = (\mathbf{mi}_i\ m_\rho(\vec{v}))}{(\mathbf{methods}\ \rho \bullet c) \xrightarrow{o_i.m(\vec{v})?} (\mathbf{methods}\ \rho \bullet c'[\eta \backslash d])} \tag{46}$$

The transformations are straightforward and maintain the structure of the original rules.

The general format of nondeterministic rule described above is characterised by a metavariable (such as v in Rule 38 and d in Rule 38(b)) appearing in the label and in the expression or program on the right-hand side of a transition, but not in the left-hand side. Metavariables appearing only in the label (and not in the right-hand side) are already handled using placeholders in the labels, e.g., α in Rule 34.

Another, related, type of nondeterminism is generated by communication labels where, instead of the nondeterminism in the label being resolved by a context, it is resolved by a parallel process. That is, an invocation $\ell!$ matches with a responder $\ell?$, and non-matching invocation/response labels are pruned at the outermost level by Rule 25 (see, for instance, Rules 31 and 33 for an example of invocation/response labels). To avoid generating a large number of redundant response transitions, the transition relation on the response label, such as $c \xrightarrow{o.f := v?} c'$, may be transformed from a relation to a function $respond : (Cmd \times Label) \to Cmd$. This gives the following format for Rule 27 for matching.

$$\frac{c_1 \xrightarrow{\ell!} c_1' \quad respond(c_2, \ell) = c_2'}{c_1 \parallel c_2 \xrightarrow{\tau} c_1' \parallel c_2'}$$

This style eliminates nondeterminism on the responder-side; any nondeterminism on the invoker-side may be eliminated using placeholders, as above. The transformation from the relation to the function $respond$ (a Maude $equation$) is straightforward.

The translation of the rules to a tool such as Maude (or related tools for simulation, e.g., [LF08, PF07]) allows the (transformed) operational semantics to output the trace of a particular program. This gives a more concrete method for demonstrating the execution of a program. Maude also provides a facility for checking soundness (invariants) and liveness properties, and this may be used to verify aspects of particular programs; we consider formal comparison of programs below.

10.2. Verification

One of the main applications of language formalisations is analysis of that language in general, and proving properties of specific programs (model checking). Below we give a set of general conditions under which different underlying data representations in a class give the same observable behaviour to some client of that class, under the usual assumption that the interfaces (the domain of the set of methods) are the same and that fields are protected from direct access. This is therefore a type of *data refinement* [HHS86, Jon90, dRE01]. An advantage of structuring the abstract syntax according to relevant conceptual processes (classes, objects, and methods) is that many of the conditions are immediately established by term homomorphisms.

We use the operational semantics as the basis for verification with respect to (weak) trace equivalence [Mil89, vG01].[12]

Definition 47 (Weak trace equivalence). Commands c and d are *trace equivalent*, written $c \overset{\mathrm{T}}{\approx} d$, if, for all traces ℓs,

$$c \overset{\ell s}{\Longrightarrow} \quad \text{if and only if} \quad d \overset{\ell s}{\Longrightarrow} \tag{48}$$

The notation '$c \overset{\ell s}{\Longrightarrow}$' is shorthand for $(\exists\, c' \bullet c \overset{\ell s}{\Longrightarrow} c')$. Recalling that the notation $c \overset{\ell s}{\Longrightarrow} c'$ may include silent steps interleaved with the labels of ℓs, the definition states that $c \overset{\mathrm{T}}{\approx} d$ if any sequence of non-τ steps can be matched between them. We note that $\mathbf{nil} \overset{\mathrm{T}}{\approx} \mathbf{nil}$, and that if c and d are terminating commands (i.e., every execution is finite and ends in $\mathbf{nil}$),

$$(c \overset{\ell s}{\Longrightarrow} \mathbf{nil} \quad \Longleftrightarrow \quad d \overset{\ell s}{\Longrightarrow} \mathbf{nil}) \Rightarrow c \overset{\mathrm{T}}{\approx} d.$$

We now consider using trace equivalence for replacing a class $\mathcal{C}$ with another class $\mathcal{D}$ that uses a different (perhaps more efficient) representation of the data, as encapsulated in the object-level fields. Given fields σ_c and σ_d of objects of classes $\mathcal{C}$ and $\mathcal{D}$, respectively, we write $\sigma_c \overset{\mathrm{S}}{\approx} \sigma_d$ if the values are related (this relation is sometimes known as abstraction function, simulation relation, or coupling invariant).

We now build towards a set of constraints on general class and object structures that ensures trace equivalence. We first define equivalence on objects, of the standard form (15).

[12] Stronger notions (such as (weak) bisimulation) are not required since the language is *determinant* [Mil89].

Definition 49 (Object equivalence).

$$(\mathbf{obj}\ o(\sigma_c)_k \bullet c) \overset{\text{O}}{\approx} (\mathbf{obj}\ o(\sigma_d)_k \bullet d) \quad \Longleftrightarrow \quad \sigma_c \overset{\text{S}}{\approx} \sigma_d \wedge c \overset{\text{M}}{\approx} d$$

Thus, two objects are equivalent if they have the same reference and structure, their fields are related by $\overset{\text{S}}{\approx}$, and the corresponding method instances are related via $\overset{\text{M}}{\approx}$, defined below.

Definition 50 (Method instance equivalence). Commands c and d are equivalent, written $c \overset{\text{M}}{\approx} d$, if they always terminate, do not contain calls to methods that modify the fields of the local object, and if the following holds for all σ_c and σ_d such that $\sigma_c \overset{\text{S}}{\approx} \sigma_d$.

$$(\mathbf{state}\ \sigma_c \bullet c) \overset{\text{T}}{\approx} (\mathbf{state}\ \sigma_d \bullet d) \tag{51}$$

$$(\mathbf{state}\ \sigma_c \bullet c) \overset{\ell s}{\Longrightarrow} \sigma_c' \wedge (\mathbf{state}\ \sigma_d \bullet d) \overset{\ell s}{\Longrightarrow} \sigma_d' \quad \Rightarrow \quad \sigma_c' \overset{\text{S}}{\approx} \sigma_d' \tag{52}$$

We have used the abbreviation

$$(\mathbf{state}\ \sigma \bullet c) \overset{\ell s}{\Longrightarrow} \sigma' \mathrel{\widehat{=}} (\mathbf{state}\ \sigma \bullet c) \overset{\ell s}{\Longrightarrow} (\mathbf{state}\ \sigma' \bullet \mathbf{nil})$$

Property (51) ensures that any returned values, or communication with the environment, will be indistinguishable between the classes, while (52) ensures that the operations on the local fields maintain the coupling invariant.

For brevity Definition 50 includes the constraint that method bodies do not make calls to methods that modify the fields of the local object; to relax this constraint the standard approach of treating method calls as a shorthand for their specification would be employed (for example, if m is a method that updates local field f, a call $\mathbf{this}.m()$ within c would be replaced by $f := ..$).

We now define class equivalence.

Definition 53 (Class equivalence) For classes $\mathcal{C}$ and $\mathcal{D}$ of the form below,

$$\mathcal{C} \mathrel{\widehat{=}} \mathbf{class}\ \mathrm{C}(\emptyset, \rho_c, \gamma_c)_n \bullet \|_{i<n}\ \mathcal{O}_i^c \qquad \textit{and} \qquad \mathcal{D} \mathrel{\widehat{=}} \mathbf{class}\ \mathrm{C}(\emptyset, \rho_d, \gamma_d)_n \bullet \|_{i<n}\ \mathcal{O}_i^d \tag{54}$$

where γ_c and γ_d are of the (standard) form,

$$\gamma_c \mathrel{\widehat{=}} (\lambda\, n \bullet \mathbf{obj}\ \mathrm{C}_n(\sigma_c^{init})_0 \bullet \mathbf{nil}) \qquad \textit{and} \qquad \gamma_d \mathrel{\widehat{=}} (\lambda\, n \bullet \mathbf{obj}\ \mathrm{C}_n(\sigma_d^{init})_0 \bullet \mathbf{nil}) \tag{55}$$

then $\mathcal{C} \overset{\text{C}}{\approx} \mathcal{D}$ iff

$$\sigma_c^{init} \overset{\text{S}}{\approx} \sigma_d^{init} \tag{56}$$

$$\forall\, i < n \bullet \mathcal{O}_i^c \overset{\text{O}}{\approx} \mathcal{O}_i^d \tag{57}$$

$$\mathrm{dom}(\rho_c) = \mathrm{dom}(\rho_d) \tag{58}$$

$$\forall\, m \in \mathrm{dom}(\rho_c), \vec{w} \bullet m_{\rho_c}(\vec{w}) \overset{\text{M}}{\approx} m_{\rho_d}(\vec{w}) \tag{59}$$

Thus two classes are equivalent if the initial states of new objects are related by the coupling invariant, the objects currently executing are related, the interfaces are the same, and each method gives the same observable trace and changes the fields so that the coupling invariant is maintained.

Note that as an immediate consequence of the definitions of γ_c and γ_d, and from (56), we have the following for all n (recall Definition 49).

$$\gamma_c(n) \overset{\text{O}}{\approx} \gamma_d(n) \tag{60}$$

In giving the general form of the classes in (54), we have let $(\|_{i<n}\ c_i)$ abbreviate $(c_1 \parallel \ldots \parallel c_{n-1})$. If $n = 1$ the command is equivalent to **nil**. Condition (57) is more general than need be, since classes start with no objects, i.e., $n = 1$, and hence the quantification is vacuously true, however taking the more general structure helps in subsequent proofs. Condition (58) will typically hold by definition and not require a separate proof. Note that in (59) we implicitly assume the quantification of $\vec{w}$ is over well-formed actual parameters only.

In the sequel, to minimise possible confusion arising from the symbol '$\Rightarrow$' being used for both a transition of 0 or more internal steps and logical implication, we will use '$\overset{\tau}{\Longrightarrow}$' to represent the former.

We now state a general theorem for showing trace equivalence.

Theorem 61 *In a single-threaded context, if* $\mathcal{C} \overset{\text{C}}{\approx} \mathcal{D}$ *then* $\mathcal{C} \overset{\text{T}}{\approx} \mathcal{D}$.

Proof. We prove by case analysis on the labels that $\mathcal{C}$ may generate. We first note that if an object in $\mathcal{C}$ has a (non-**nil**) method instance, c, in its workspace, then c runs until it terminates. If no such object exists, then $\mathcal{C}$ is free to respond to three types of requests: a creation of a new object, an invocation of a method in an existing object, or an instance-of query (in a multi-threaded environment, locks on monitors may also be requested). If we let ℓs be a trace corresponding to the execution of a method instance inside an object, and r be a label of the form $\mathsf{new}(\text{C})n?$, $o_i.m(\vec{v})?$, or $o.\mathsf{instOf}(\text{C})?$, then a trace of $\mathcal{C}$ is a nonterminating trace of the form $(r^+ \quad \ell s)^*$, where r^+ represents a sequence of 1 or more instances of r and r^* a sequence of 0 or more instances.

Therefore to prove trace equivalence it suffices to show that after each response r or execution of a method instance ℓs that the resulting classes preserve class equivalence. More formally, we prove the following, where rs is either a singleton trace containing an instance of label type r, or a trace corresponding to the execution of a method instance.

$$\mathcal{C} \overset{rs}{\Longrightarrow} \quad \Leftrightarrow \quad \mathcal{D} \overset{rs}{\Longrightarrow} \tag{62}$$

$$\mathcal{C} \overset{rs}{\Longrightarrow} \mathcal{C}' \wedge \mathcal{D} \overset{rs}{\Longrightarrow} \mathcal{D}' \quad \Rightarrow \quad \mathcal{C}' \overset{\text{C}}{\approx} \mathcal{D}' \tag{63}$$

Together these conditions show that any two related classes work in tandem, and that the resulting classes are also related. This gives trace equivalence, since any trace of $\mathcal{C}$ and $\mathcal{D}$ may be subdivided into identical subtraces which preserve equivalence.

We now prove (62) and (63) by case analysis on rs, in all cases implicitly assuming $\mathcal{C} \overset{\text{C}}{\approx} \mathcal{D}$. We first note that (62) for the r cases is trivial, since a class may always respond to such requests ($\mathcal{C} \xrightarrow{r}$ is always true). Therefore below we show just (63) for those cases.

New object. Take the case where $rs = \mathsf{new}(\text{C})n?$. Using Rule 27 gives the following transitions.

$$\mathcal{C} \xrightarrow{\mathsf{new}(\text{C})n?} \mathcal{C}^+\gamma_c(n) \qquad \wedge \qquad \mathcal{D} \xrightarrow{\mathsf{new}(\text{C})n?} \mathcal{D}^+\gamma_d(n)$$

By (60) we have $\gamma_c(n) \overset{\text{O}}{\approx} \gamma_d(n)$ and hence $\mathcal{C}^+\gamma_c(n) \overset{\text{C}}{\approx} \mathcal{D}^+\gamma_d(n)$, therefore equation (63) holds.

New method instance. Take the case where $rs = o_i.m(\vec{v})$ (for $m \in \mathrm{dom}(\rho_1)$). Using Rule 23 gives the following transitions (where o abbreviates object reference C_i).

$$\mathcal{C}[\mathcal{O}_i^c] \xrightarrow{o_k.m(\vec{w})?} \mathcal{C}[\mathcal{O}_i^{c+}c] \qquad \textit{and} \qquad \mathcal{D}[\mathcal{O}_i^d] \xrightarrow{o_k.m(\vec{w})?} \mathcal{D}[\mathcal{O}_i^{d+}d]$$

where $c = (\mathbf{mi}_k \;\; m_{\rho_1}(\vec{w}))$ and $d = (\mathbf{mi}_k \;\; m_{\rho_2}(\vec{w}))$. From (59), we have $m_{\rho_1}(\vec{w}) \overset{\text{M}}{\approx} m_{\rho_2}(\vec{w})$, and from Rules 40 and 41 for method instances we can deduce $c \overset{\text{M}}{\approx} d$. Thus the addition of a new method preserves $\mathcal{O}_i^{c+}c \overset{\text{O}}{\approx} \mathcal{O}_i^{d+}d$, and hence $\mathcal{C}[\mathcal{O}_i^{c+}c] \overset{\text{C}}{\approx} \mathcal{D}[\mathcal{O}_i^{d+}d]$. If subclassing and polymorphism were employed the class-specific method calls would be dealt with similarly.

Instance-of responses. Rule 49 does not change the class and hence trivially satisfies (63).

Method instances. The crux of the proof lies in showing that the method bodies maintain the class equivalence homomorphism (and therefore, importantly, method calls always return the same values).

Recall that each object in $\mathcal{C}$ and $\mathcal{D}$ satisfies $\mathcal{O}_i^c \overset{\text{O}}{\approx} \mathcal{O}_i^d$, and that corresponding method instances in those objects satisfy $c \overset{\text{M}}{\approx} d$. This means that, although c and d generate different traces due to operating on different variables, they produce the same trace when executed in the local state provided by their containing objects (the operations on local fields become internal τ steps). The remaining labels, including invocation labels accessing other object's fields and methods, exceptions, etc., are also precisely the same. Thus, at the object (and class) level, the traces generated are the same, and preserve the class equivalence.

More formally, we pick some object $\mathcal{O}_i^c$ in the workspace of $\mathcal{C}$ (and a corresponding $\mathcal{O}_i^d$) executing method instance c (respectively d). By assumption $c \overset{\text{M}}{\approx} d$. This means that executing c and d until termination in their respective object contexts, which include the fields, generates the same trace (51). In addition, the new values for the the fields preserve $\overset{\text{S}}{\approx}$ from (52). Thus executing method instances that satisfy (59) preserves class equivalence.

Let $\mathcal{O}_i^c[c]$ ($\mathcal{O}_i^d[d]$) be the object executing its method instance. Assumption (59) immediately implies

$$\mathcal{C}[\mathcal{O}_i^c[c]] \overset{\ell s}{\Longrightarrow} \qquad \Leftrightarrow \qquad \mathcal{D}[\mathcal{O}_i^d[d]] \overset{\ell s}{\Longrightarrow}$$

which gives (62). From (52) We have

$$\mathcal{C}[\mathcal{O}_i^c[c]] \stackrel{\ell s}{\Longrightarrow} \mathcal{C}[\mathcal{O}_i^{c\prime}[\mathbf{nil}]] \quad \Leftrightarrow \quad \mathcal{D}[\mathcal{O}_i^d[d]] \stackrel{\ell s}{\Longrightarrow} \mathcal{D}[\mathcal{O}_i^{d\prime}[\mathbf{nil}]]$$

where the fields of $\mathcal{O}_i^{c\prime}$ and $\mathcal{O}_i^{d\prime}$ are related via $\overset{\text{S}}{\approx}$, and hence $\mathcal{O}_i^{c\prime} \overset{\text{O}}{\approx} \mathcal{O}_i^{d\prime}$. Thus, we have $\mathcal{C}[\mathcal{O}_i^{c\prime}[\mathbf{nil}]] \overset{\text{C}}{\approx} \mathcal{D}[\mathcal{O}_i^{d\prime}[\mathbf{nil}]]$, and therefore (63) holds. □

In the above conditions and proof we have for simplicity assumed a single-threaded environment. A verification theory handling concurrency is beyond the scope of this paper, but we hope to have demonstrated how using traces may provide the foundation for proving more complex properties. Interactions between threads and locks are specified through labels in the traces, and these may be unified with state-based aspects as outlined above. Interference may be handled using notions such as rely-guarantee reasoning [Jon83b, Jon83a, CJ07].

We now provide an example of Theorem 61. Consider the following classes. For readability we let '$m() \mathrel{\widehat{=}} c$' abbreviate the pair $m \mapsto c$ and similarly '$m(x) \mathrel{\widehat{=}} c$' abbreviate the pair $m \mapsto (\lambda\, v \bullet \mathbf{state}\ \{x \mapsto v\} \bullet c)$.

$$\begin{array}{ll}
\mathcal{C} \mathrel{\widehat{=}} \mathbf{class}\ \mathrm{C}(\emptyset, \rho_1, \gamma_1)_1 \bullet \mathbf{nil} & \mathcal{D} \mathrel{\widehat{=}} \mathbf{class}\ \mathrm{C}(\emptyset, \rho_2, \gamma_2)_1 \bullet \mathbf{nil} \\
\gamma_1 \mathrel{\widehat{=}} (\lambda\, n \bullet \mathbf{obj}\ \mathrm{C}_n(\{l \mapsto \langle\rangle\})_0 \bullet \mathbf{nil}) & \gamma_2 \mathrel{\widehat{=}} (\lambda\, n \bullet \mathbf{obj}\ \mathrm{C}_n(\{s \mapsto 0\})_0 \bullet \mathbf{nil}) \\
\rho_1 \mathrel{\widehat{=}} \left\{ \begin{array}{rcl} init_0() & \mathrel{\widehat{=}} & \mathbf{nil} \\ add(x) & \mathrel{\widehat{=}} & l := l \frown x \\ getsum() & \mathrel{\widehat{=}} & \mathbf{return}\ \Sigma\, l \end{array} \right\} & \rho_2 \mathrel{\widehat{=}} \left\{ \begin{array}{rcl} init_0() & \mathrel{\widehat{=}} & \mathbf{nil} \\ add(x) & \mathrel{\widehat{=}} & s \mathrel{+}= x \\ getsum() & \mathrel{\widehat{=}} & \mathbf{return}\ s \end{array} \right\}
\end{array} \tag{64}$$

An object of class $\mathcal{C}$ has a field l (a list of numbers that is initially empty), an empty initialiser $init_0$, a method add for adding a new number to the list, and a method $getsum$ for returning the current total of numbers in the list ($\Sigma\, l$). An object of class $\mathcal{C}$ is generated by the constructor γ_1, the methods are collected in ρ_1 (all of which are object-level), and there are no class-level fields. Class $\mathcal{D}$ is similar except that objects use a field s instead of l, which keeps track of the running total of values added. This is a more efficient version of the class, assuming that $getsum$ is called regularly. We assume that all calls $add(e)$ are well-formed, that is, that e evaluates to a number.

Theorem 65 *Classes $\mathcal{C}$ and $\mathcal{D}$, defined in (64), are trace equivalent.*

Proof. By Theorem 61 we need only show that $\mathcal{C} \overset{\text{C}}{\approx} \mathcal{D}$. Firstly we note that the classes and constructors are of the forms given in (54) and (55). We next define the relationship between fields.[13]

$$\{l \mapsto \vec{v}\} \overset{\text{S}}{\approx} \{s \mapsto v\} \quad \Leftrightarrow \quad \Sigma\, \vec{v} = v \tag{66}$$

We immediately see that condition (56) is satisfied since $\Sigma\, \langle\rangle = 0$. Condition (57) is trivially satisfied since $n = 1$ and the workspaces are empty. Condition (58) is also trivially satisfied by definition. This leaves (59) as the only non-trivial requirement, which we prove below for each $m \in \{init_0, add, getsum\}$ and well-formed $\vec{w}$. In all cases we assume $\Sigma\, \vec{v} = v$.

1. $init_0$. The proof of both (51) and (52) is trivial since the body of $init_0$ in both cases is **nil** (the local fields therefore preserve (66)).

   $$(\mathbf{state}\ \{l \mapsto \vec{v}\} \bullet init_{0\rho_1}()) \stackrel{\tau}{\Longrightarrow} (\mathbf{state}\ \{l \mapsto \vec{v}\} \bullet \mathbf{nil}) \quad \textit{and}$$

   $$(\mathbf{state}\ \{s \mapsto v\} \bullet init_{0\rho_2}()) \stackrel{\tau}{\Longrightarrow} (\mathbf{state}\ \{s \mapsto v\} \bullet \mathbf{nil})$$

2. add. Recall that $add_{\rho_2}(w) = (\mathbf{state}\ \{x \mapsto w\} \bullet s \mathrel{+}= x)$. Letting $v' = v + w$ (the premise for Rule 9(c)) and recalling Rule 18, by straightforward application of the rules for executing code we get a trace of the following form.

   $$(s \mathrel{+}= x) \xrightarrow{s=v} (s := v + x) \xrightarrow{x=w} (s := v + w) \longrightarrow (s := v') \xrightarrow{s := v'} \mathbf{nil}$$

[13] Given a predicate e, if we let $e[\sigma]$ represent substituting the variables in the domain of σ by their corresponding values, we may define instead

$$\sigma_c \overset{\text{S}}{\approx} \sigma_d \Leftrightarrow (\Sigma\, l = s)[\sigma_c][\sigma_d].$$

This is a slightly easier to read format for the coupling invariant; the format may be used provided the field names of each class are distinct.

A similar trace is generated for $add_{\rho_1}(w)$, but instead operating on l. Clearly, those traces are not equivalent, however, when executed inside the local state for the method parameter x, and the object fields s and l, all labels are hidden. This satisfies (51).
To show (52) we observe

$$(\mathbf{state}\ \{l \mapsto \vec{v}\} \bullet add_{\rho_1}(w)) \stackrel{\tau}{\Longrightarrow} (\mathbf{state}\ \{l \mapsto \vec{v}'\} \bullet \mathbf{nil}) \quad \textit{and}$$

$$(\mathbf{state}\ \{s \mapsto v\} \bullet add_{\rho_2}(w)) \stackrel{\tau}{\Longrightarrow} (\mathbf{state}\ \{s \mapsto v'\} \bullet \mathbf{nil})$$

where $\vec{v}' = \vec{v} \frown w$. Assuming $\Sigma\vec{v} = v$, we have $(\Sigma\vec{v}') = (\Sigma(\vec{v} \frown w)) = (\Sigma\vec{v} + w) = (v + w) = v'$, and so (66) is maintained by the execution of add.

3. $getsum$. As with add, by straightforward application of the rules for executing code, we have the following traces.

$$(\mathbf{state}\ \{l \mapsto \vec{v}\} \bullet getsum_{\rho_1}()) \stackrel{\alpha_i.\mathsf{return}(v)!}{\Longrightarrow} (\mathbf{state}\ \{l \mapsto \vec{v}\} \bullet \mathbf{nil}) \qquad \textit{and}$$

$$(\mathbf{state}\ \{s \mapsto v\} \bullet getsum_{\rho_2}()) \stackrel{\alpha_i.\mathsf{return}(v)!}{\Longrightarrow} (\mathbf{state}\ \{s \mapsto v\} \bullet \mathbf{nil})$$

This method contains an explicit return label in the trace, but in both cases the returned value is the same and hence satisfies (51). The fields are not modified and hence (52) holds trivially. □

Having proved trace equivalence, we may deduce that a program using the implementation class $\mathcal{C}$ in (64) gives an equivalent behaviour to that using class $\mathcal{D}$. For instance, consider the following $Main$ process, which interacts with the environment to obtain a number to add to the list.

$$Main \mathrel{\widehat{=}} o := \mathbf{new}\ \mathrm{C}()\ ;\ o.add(\mathbf{input})\ ;\ \mathbf{output}\ o.getsum() \tag{67}$$

Since $\mathcal{C} \stackrel{\mathrm{T}}{\approx} \mathcal{D}$, we also have $(Main \parallel \mathcal{C}) \stackrel{\mathrm{T}}{\approx} (Main \parallel \mathcal{D})$, with interactions determined by Rule 27. At the environment level the only visible labels are input and output, and the trace of the composition is always of the form 'input (v) output (v)', regardless of whether $\mathcal{C}$ or $\mathcal{D}$ is used.

Note that in Definition 53 we set the class-level fields to empty. This was purely to avoid distraction; if the class-level fields are the same, trace equivalence will follow as long as they are manipulated the same way in each method instance. If the representation of the class-level fields also changes, that needs to be reflected in the definition of $\stackrel{\mathrm{S}}{\approx}$, and the class-level fields added to the context in Definition 50.

In the proofs we assumed that the fields could not be accessed externally to the class. If this were not the case, then a client of the object could distinguish them through the exposed types (and names) of the fields. Instead of simply assuming the fields are not accessed, this can be enforced semantically with commands such as $(\mathbf{protect}_F\ c)$, defined by the following rule.

$$\frac{c \xrightarrow{o.f=v?} c' \quad f \notin F}{(\mathbf{protect}_F\ c) \xrightarrow{o.f=v?} (\mathbf{protect}_F\ c')}$$

The command type $(\mathbf{protect}_F\ c)$ prevents testing the value of a field f if it is in the protected set F. This command may be used to selectively specify the set of protected fields (those that are likely to have their representation changed) within the object. Similar rules can be used to prevent updating fields, and to protect methods of an object.

More general properties of the language, such as distributivity and commutativity of operators, may be proved with respect to general theorems about SOS formats with singleton configurations [TP97, GMR06, JGH11, Kli11]. More language-specific properties may also be simpler to both express and prove if they are with respect to an individual command (e.g., class invariants [LM05, DJO08, Log09]), since the context is encapsulated and the interface specified by events. Many frameworks developed for formal verification of imperative programming languages (e.g., [Spi92, Mor94, dRE01, dRdBH+01]) have been adapted for object-oriented programs (e.g., [CKRW99, Fru10, ÁdBdRS08, dB09, BO08, AdBOdG12, DDJO12]). In the verification above we have emphasised compositional reasoning [Jon03b] using the trace-based semantics, and this could be explored further with respect to the notions of modularity defined by Müller et al. [MPHL06]. Alternatively, different program structures, such as those used in the *box model* [PHS06], may be more convenient for some types of properties than the class-based structure used above.

11. Related work

There has been much research into the semantics of object-oriented programming languages; overviews appear in several survey papers and collected works [AF99, HM01, Nip03, LLM07]. Much of this work has focused on developing a semantics that is amenable to analysis, and hence the emphasis has been less on the style of the semantics and more on verification techniques and their automation. In contrast, this paper has emphasised expressing core concepts with several goals in mind: a term structure that reflects the class hierarchy; a separation of concerns when dealing with core concepts; straightforward composition of basic rules into more powerful rules; a flexible communication scheme based on process algebra theory; and the semantics of local variables (such as those in methods, or object-level fields and class-level fields) is defined so that reasoning about them may be performed locally. On a technical level, these goals were realised by keeping contextual information (method definitions, local variables, constructors, etc.) within the program term structure, and using labelled transitions to represent both communication events and state modifications. More powerful rules can be derived from the basic rules to give the semantics of individual processes, including method instances, objects, and classes. These may in turn be composed to give the behaviour of the system as a set of interacting processes, as described in Sect. 2.2.

The majority of approaches in the literature do not have an abstract syntax that follows the class hierarchy, and few treat objects and classes as encapsulated processes. Typically, related information from each object, such as fields or methods, is grouped together in a store, and hence the state of a particular object or class may be inferred by extracting the relevant information from each store. Therefore it is not possible in such frameworks to give high-level, structural rules such as those in Sect. 2.2. Usually a single call stack is defined (or a set of stacks for multithreading), with successive method calls, and the relevant environment, pushed on to the stack, and popped after completion. In contrast, in our framework active method instances are distributed across the relevant objects.

Language formalisation is an ongoing research concern and is often tackled through either denotational semantics [Rei95, AFL99, SSL08] or operational semantics [MTM97, MMT08, MF08, ER10]. We have provided an operational semantics because, as noted by Jones [Jon03a], this style lends itself to concurrency more easily than denotational semantics. Below we restrict comparison to work which defines operational semantics for object-oriented programming language features. With the notable exception of the work of Börger et al., discussed below, our presentation appears to cover more object-oriented concepts than those surveyed.

The work was initially inspired by Jones' semantics for an object-oriented language [Jon07], where objects are processes encapsulating their own state and which execute method calls locally, and classes define the relevant methods. However, in Jones' work there is no inheritance, and objects are not nested within their class. All parallelism is at the top level, and managed through a set of objects rather than through the process-algebraic parallel operator ('$\|$'). The style results in verbose, though straightforward, rules. For the purposes of clarity and writing correct code, Jones enforces several restrictions, particularly on parallelism, which could be straightforwardly accommodated in our semantics.

Börger, Stark, et al. [SSB01, BFGS05] have defined comprehensive formal semantics for the Java and C# programming languages. The treatment is based on abstract state machines (ASMs) [BS03], and covers all of the constructs given here as well as other language features, in particular a richer set of operations on threads. The approach has been used successfully to develop analysis techniques [Fru04, Stä05, Fru10]. A program is dynamically structured as a stack of method calls, and in the multi-threaded case, as a set of stacks. New methods within a thread are placed on the corresponding stack, and yield their return when completed (and popped). The call stack is separate to other relevant information such as method definitions, globals, locals, and locks. As a consequence, the semantics does not allow easy decomposition into individual object behaviour—method instances of different objects are interleaved on the stack. In our semantics, an object executes its own method instances locally, and interacts with other objects through explicit communication (invoking and returning from methods). Arguably, our framework is more abstract, since it is in keeping with a traditional process-algebraic view of concurrency and communication. On the other hand, the stack-based method is closer to a real implementation, and hence may be better suited to language implementation details than the framework presented here.

Abadi and Cardelli [AC95, AC96] provide a comprehensive treatment of the underlying concept of objects, classes, and object-oriented programming. Their framework is particularly rigorous in defining the meaning of subclassing, subtyping and interfaces, and is extended to include higher-order programming features. The meaning of a (concrete) programming language is defined by translating it to the abstract theory, for which they provide

a big-step execution semantics. Their treatment is more abstract than that given here, where we have focused on the definition of programming language features, and define our rules directly on a familiar programming syntax. Their language is *object-based*, which means that, instead of calling constructors of a class, new objects are generated by cloning existing objects. They show that this is more flexible than *class-based* languages, such as that presented here. The distinction leads them to introduce the powerful *method update* primitive, where the methods of an individual object can be dynamically altered, and which generalises field update. We have focused on defining the semantics of an object-oriented programming language like Java, and hence have kept the structure of a class-based framework and have not introduced the method update command. Because our method declarations are mappings from identifiers to parameterised commands, rules can be defined for updating them using a similar pattern to that for updating fields. By then moving the method declarations from the class level to the object level, our framework can be extended to adopt Abadi and Cardelli's more abstract approach.

Klein and Nipkow [KN06] give an operational semantics for a subset of Java which which does not include subclassing or multithreading (however see Lochbihler [Loc10] for a multithreaded extension of the framework). All rules, and several properties of the language, were encoded in the theorem prover Isabelle/HOL [NPW02]; furthermore, they define the Java virtual machine, the bytecode verifier, and a compiler. Ábrahám et al. [ÁdBdRS03, ÁdBdRS08] give operational semantics for subsets of Java that include exceptions and multithreading. The languages are less expressive than ours in that subclassing is not tackled, and expressions are more restricted, e.g., method calls cannot appear in expressions. de Boer [dB09] treats a similar subset of object-oriented programming features to that of [ÁdBdRS08], but in his framework the threads are the top-level processes (each process is a thread). In contrast, our threads are spread throughout the structure of the program, and are not represented by any one process. de Boer's treatment simplifies thread-based verification conditions. Jeffrey and Rathke [JR05] present a trace-based operational semantics for a subset of Java, where the traces include method calls and constructor invocations, but their language does not directly support multithreading, local variables, or class-level fields.

The *box model* [PHS06, PHS07, BDDL+10] is a framework for describing the cooperation of collections of objects, which may be unrelated by the class hierarchy. In this setting the focus is not on the structure of the class hierarchy, but on relationships between objects that have access to each other, for instance, a group of objects that cooperate to implement a particular function. Due to encapsulation and the potentially nested nature of boxes, it may be possible to adapt the semantics in this paper to apply to the very different program structure of the box model.

In earlier work with Hayes [CH09], we investigated the same technique of using labels and contexts to add state, guards and assignments to the process algebra CSP [Hoa85], and defined the meaning of a sequential programming language with procedure calls and recursion [CH11]. Owens [Owe08] and Abadi and Harris [AH09] used variable test and update labels for simplifying their operational semantics. Brookes [Bro07] uses similar basic labels in a denotational semantics for a concurrent language, and in particular for defining the semantics of local variables. The work we present here goes beyond these papers in defining interactions between processes on otherwise inaccessible local contexts, in an object-oriented setting.

12. Conclusions

In this paper we have presented an operational semantics for a set of core object-oriented programming concepts. The abstract syntax of a program conforms to the class hierarchy [DGC95]. The semantics is based around the promotion of labels and their interaction with the context. The language uses a CCS-based semantics [Mil82] for interprocess communication. With the notable exception of the work of Börger et al. [SSB01, BFGS05], the language we have presented covers more object-oriented features than most formal treatments in the literature. We conjecture that this is because for large and complex languages standard approaches to operational semantics become unwieldy, with the rules becoming cluttered by the amount of information required for execution. This is less of a problem in our framework because information is stored as context in the language, and the behaviour of a process can be observed through its traces (the sequence of labels it generates). This is less convenient in semantics where information about the entire program, i.e., the state and environment, appears in every rule.

Trace equivalence emerges naturally from the semantics as a verification technique due to several reasons. Firstly, operations on local variables are hidden, which handles the change in state space from one class to another, and the communication through method calls is handled using synchronisation; and secondly, because the processes involved in the proof (classes, objects, and methods) are encapsulated we can partition the proof at different levels. In practice, many aspects of the proof are satisfied trivially by the structure of the processes.

Following Börger et al. [SSB01, BFGS05], we presented the semantics incrementally. The rule framework is defined so that language extensions do not typically require the redefining of earlier rules—the simple rules remain simple, regardless of the complexity of the language. The exception was when introducing threads: two new rules replaced earlier non-threaded rules, although in both cases the structure of the rule remained the same. Although a minor issue in this paper, maintaining modularity of the rules [Mos02] may be addressed by using a record structure in the label and adopting the style developed by Mosses [Mos04b, Mos04a, MN09].

The semantics we have presented here followed from a general principle of keeping the syntax familiar to programmers, that is, as close as possible to concrete syntax. We also took a "separation of concerns" approach; for instance, separating the fields and methods of an object from the reference for the object. An advantage of this approach is the syntax and semantics for fields and methods is simpler than if the reference was not separated; the trade off is that rules are required to ensure the correct reference is used. The approach taken was always the one which conformed to a simple syntax and separation of concerns, even if some rules were therefore more complex. One of the major decisions was to keep method definitions in the class, rather than in the object. This decision was taken because it seems more intuitive for there to be only one copy of the definitions (as in a program), and so that the semantics of dynamic dispatch are constructed dynamically. However, it would be valid to define object constructors so that each new object is given its own copy of the correct methods, collecting all of the inherited methods and renaming those that are overridden in subclasses. This approach places a greater burden on the translation from concrete syntax to abstract syntax, and to some extent loses the nested nature of method definitions; on the other hand, it would simplify the semantics of method calls, and class-specific method calls, to associate method definitions directly with the object, and this approach may be better for some applications.

Most real object-oriented programming languages contain more features than those presented here, for instance, Java has a richer set of thread operations, polymorphism, class initialisation, etc. We have focused on a set of core aspects of object-orient languages (objects, classes, fields, methods, constructors, inheritance and synchronisation), with the intention that the framework is flexible enough to be extended to incorporate extra features. A notable language feature which is inherently not compatible with the framework in its current state is multiple inheritance [WNST06, BCV09, DP11], which would require a term (object) to be a subterm of two distinct parent terms, something that is not directly achievable.

Acknowledgements

The author thanks Ian Hayes for early encouragement with the work, in particular to pursue the "separation of concerns" approach. The author also thanks the anonymous referees for many helpful comments.

A. More on exceptions

A.1. Null reference

To avoid distraction, in the paper we assume that all object references are defined. However it is common to use a special value, say null, for an undefined reference. The rules required to throw an exception when a null reference is encountered are of the following form.

$$\mathsf{null}.f \xrightarrow{\mathsf{throw}(nullref)} \mathsf{null} \qquad \mathsf{null}.f := v \xrightarrow{\mathsf{throw}(nullref)} \mathbf{nil} \qquad \mathsf{null}.m(v) \xrightarrow{\mathsf{throw}(nullref)} \mathbf{nil} \qquad \ldots$$

A.2. Finally clauses

Based on the Java language specification [AGH00, Sect. 14.20.2] (see also, e.g., [Jac01]), the '*try*/*catch*/*finally*' command may be decomposed into a *try*/*catch* command inside the special *finally* command, i.e., **try** { `c1` } **catch** `(exc)` { `c2` } **finally** { `c3` } becomes

(**try** c_1 **catch** (exc) c_2) **finally** c_3.

The semantics of the *finally* command is given below.

$$\frac{\text{Command } (c_1 \textbf{ finally } c_2) \text{ promotes } \ell \text{ through } c_1 \text{ for } \ell \text{ not equal to } \mathsf{throw}(exc) \qquad c_1 \xrightarrow{\mathsf{throw}(exc)} c_1'}{(c_1 \textbf{ finally } c_2) \longrightarrow (c_2 \,;\, \textbf{throw } exc)} \qquad (\textbf{nil finally } c_2) \longrightarrow c_2$$

Hence, a finally clause postpones an exception which has been thrown (uncaught) by c_1 until after c_2 has executed. If c_2 throws an exception, exc_2, then the original thrown exception is lost.

A.3. Exceptions as objects

In the body of the paper we treated exceptions as simple values. A richer, and more realistic, treatment, is to define exceptions as classes, and for the **throw** command to throw a reference to an exception object. This allows information to be stored with the exception, which can be accessed in the catch clause.

The expression of a **throw** command is evaluated to a reference before being thrown, as given by the following rules.

$$(\text{Command } (\textbf{throw } e) \text{ promotes } \ell \text{ through } e) \qquad (\textbf{throw } o \xrightarrow{\mathsf{throw}(o)} \textbf{nil})$$

To catch an exception object, we let *catch* commands specify a class E for the exception being caught, (**try** c_1 **catch** (E exc) c_2). This command catches an object reference o if it is an object of class E, which becomes the value of local variable exc in the command c_2, and otherwise throws the exception.

$$\frac{c_1 \xrightarrow{\mathsf{throw}(o)} c_1'}{(\textbf{try } c_1 \textbf{ catch } (\text{E } exc)\ c_2) \longrightarrow \textbf{if } (o \textbf{ instOf } \text{E}) \textbf{ then } (\textbf{state } \{exc \mapsto o\} \bullet c_2) \textbf{ else } (\textbf{throw } o)}$$

References

[AC95] Abadi M, Cardelli L (1995) An imperative object calculus. In: Mosses P, Nielsen M, Schwartzbach M (eds) Theory and practice of software development (TAPSOFT 95), volume 915 of lecture notes in computer science. Springer, Berlin/Heidelberg, pp 469–485

[AC96] Abadi M, Cardelli L (1996) A theory of objects. Springer, Berlin

[ÁdBdRS03] Ábrahám E, de Boer FS, de Roever WP, Steffen M (2003) A compositional operational semantics for Java $_{\text{mt}}$. In: Dershowitz N (ed) Verification: theory and practice, vol 2772 of lecture notes in computer science. Springer, Berlin, pp 290–303

[ÁdBdRS08] Ábrahám E, de Boer FS, de Roever WP, Steffen M (2008) A deductive proof system for multithreaded Java with exceptions. Fundam Inform 82(4):391–463

[AdBOdG12] Apt KR, de Boer FS, Olderog E-R, de Gouw S (2012) Verification of object-oriented programs: a transformational approach. J Comput Syst Sci 78(3):823–852

[AF99] Alves-Foss J (ed) (1999) Formal syntax and semantics of Java, volume 1523 of lecture notes in computer science. Springer, Berlin

[AFL99] Alves-Foss J, Lam F (1999) Dynamic denotational semantics of Java. In: Alves-Foss J (ed) Formal syntax and semantics of Java, volume 1523 of lecture notes in computer science. Springer, Berlin/Heidelberg, pp 541–541

[AGH00] Arnold K, Gosling J, Holmes D (2000) The Java programming language, 3rd edn. Addison-Wesley, Boston

[AH09] Abadi M, Harris T (2009) Perspectives on transactional memory. In: Bravetti M, Zavattaro G (eds) Proceeding of concurrency theory (CONCUR 2009), volume 5710 of lecture notes in computer science. Springer, Berlin, pp 1–14

[BCV09] Bettini L, Capecchi S, Venneri B (2009) Dynamic overloading with copy semantics in object-oriented languages: a formal account. RAIRO Theor Inform Appl 43(03):517–565

[BDDL+10] Bettini L, Damiani F, De Luca M, Geilmann K, Schfer J (2010) A calculus for boxes and traits in a Java-like setting. In: Clarke D, Agha G (eds) Coordination models and languages, volume 6116 of lecture notes in computer science. Springer, Berlin/Heidelberg, pp 46–60

[BDMN73] Birtwistle GM, Dahl O-J, Myhrhaug B, Nygaard K (1973) SIMULA begin. Auerbach Publishers Inc, Philadelphia

[BFGS05] Börger E, Fruja NG, Gervasi V, Stärk RF (2005) A high-level modular definition of the semantics of C#. Theor Comput Sci, 336(2–3):235–284

[BK84] Bergstra JA, Klop JW (1984) Process algebra for synchronous communication. Inform Control 60(1–3):109–137

[BO08] Blanchette JC, Owe O (2008) An open system operational semantics for an object-oriented and component-based language. In: Electronic notes in theoretical computer science, 215(0):151–169, Proceedings of the 4th international workshop on formal aspects of component software (FACS 2007)

[Bro07] Brookes S (2007) A semantics for concurrent separation logic. Theor Comput Sci 375(1G3):227–270

[BS03] Börger E, Stärk RF (2003) Abstract state machines: a method for high-level system design and analysis. Springer, Berlin

[CB07] Chalub F, Braga C (2007) Maude MSOS tool. Electr. Notes Theor Comput Sci 176(4):133–146

[CDE+02] Clavel M, Duran F, Eker S, Lincoln P, Marti-Oliet N, Meseguer J, Quesada JF (2002) Maude: specification and programming in rewriting logic. Theor Comput Sci 285(2):187–243

[CH09] Colvin R, Hayes IJ (2009) CSP with hierarchical state. In: Leuschel M, Wehrheim H (eds) Integrated formal methods (IFM 2009), volume 5423 of lecture notes in computer science. Springer, Berlin, pp 118–135

[CH11] Colvin RJ, Hayes IJ (2011) Structural operational semantics through context-dependent behaviour. J Logic Algebraic Programm 80(7):392–426

[CJ07] Coleman JW, Jones CB (2007) A structural proof of the soundness of rely/guarantee rules. J Log Comput 17(4):807–841

[CKRW99] Cenciarelli P, Knapp A, Reus B, Wirsing M (1999) An event-based structural operational semantics of multi-threaded Java. In: Alves-Foss J (ed) Formal syntax and semantics of Java, volume 1523 of lecture notes in computer science. Springer, Berlin/Heidelberg, pp 157–200

[dB09] de Boer FS (2009) A shared-variable concurrency analysis of multi-threaded object-oriented programs. Theor Comput Sci 410(2–3):128–141

[DDJO12] Din CC, Dovland J, Johnsen EB, Owe O (2012) Observable behavior of distributed systems: component reasoning for concurrent objects. J Logic Algebraic Programm 81(3):227–256

[DGC95] Dean J, Grove D, Chambers C (1995) Optimization of object-oriented programs using static class hierarchy analysis. In: Tokoro M, Pareschi R (eds) European conference on object-oriented programming (ECOOP 1995), volume 952 of lecture notes in computer science. Springer, Berlin/Heidelberg, pp 77–101

[DJO08] Dovland J, Johnsen EB, Owe O (2008) Observable behavior of dynamic systems: component reasoning for concurrent objects. In: Electronic Notes in Theoretical Computer Science, 203(3):19–34. Proceedings of the Workshop on the Foundations of Interactive Computation (FInCo 2007).

[DP11] Ducournau R, Privat J (2011) Metamodeling semantics of multiple inheritance. Sci Comput Programm 76(7):555–586

[dRdBH+01] de Roever W-P, de Boer F, Hooman UHJ, Lakhnech Y, Poel M, Zwiers J (2001) Concurrency verification: introduction to compositional and noncompositional methods. Cambridge University Press, Cambridge

[dRE01] de Roever W-P, Engelhardt K (2001) Data Refinement: model-oriented proof methods and their comparison. Cambridge University Press, Cambridge

[ER10] Ellison C, Roşu G (2010) A formal semantics of C with applications. Technical Report, http://hdl.handle.net/2142/17414, University of Illinois, Illinois, November 2010

[FCMR04] Farzan A, Chen F, Meseguer J, Rosu G (2004) Formal analysis of Java programs in JavaFAN. In: Alur R, Peled D (eds) Computer aided verification, volume 3114 of lecture notes in computer science. Springer, Berlin/Heidelberg, pp 242–244

[FHN+11] Fisher J, Henzinger T, Nickovic D, Piterman N, Singh A, Vardi M (2011) Dynamic reactive modules. In: Katoen J-P, Knig B (eds) Concurrency theory (CONCUR 2011), volume 6901 of lecture notes in computer science. Springer, Berlin/Heidelberg, pp 404–418

[Fru04] Fruja NG (2004) Specification and implementation problems for C#. In: Zimmermann W, Thalheim B (eds) Abstract state machines, volume 3052 of lecture notes in computer science. Springer, Berlin, pp 127–143

[Fru10] Fruja NG (2010) Towards proving type safety of C#. Comput Lang Syst Struct 36(1):60–95

[GHJV95] Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional, Reading

[GMR06] Groote JF, Mousavi MR, Reniers MA (2006) A hierarchy of SOS rule formats. In; Electronic notes in theoretical computer science, 156(1):3–25, 2006. Proceedings of the second workshop on structural operational semantics (SOS 2005)

[GR02] Goldberg A, Robson D (2002) Smalltalk 80: the language and its implementation. Addison-Wesley, Reading

[HHS86] He J, Hoare CAR, Sanders J (1986) Data refinement refined (resume). In: Robinet B, Wilhelm R (eds) ESOP 86, volume 213 of lecture notes in computer science. Springer, Berlin/Heidelberg, pp 187–196

[HM01] Hartel PH, Moreau L (2001) Formalizing the safety of Java, the Java virtual machine, and Java card. ACM Comput Surv 33(4):517–558

[Hoa85] Hoare CAR (1985) Communicating sequential processes. Prentice-Hall, Inc., Upper Saddle River

[Jac01] Jacobs B (2001) A formalisation of Java's exception mechanism. In: Sands D (ed) Programming languages and systems, volume 2028 of lecture notes in computer science. Springer, Berlin/Heidelberg, pp 284–301

[JGH11] Jaskelioff M, Ghani N, Hutton G (2011) Modularity and implementation of mathematical operational semantics. In: Electronic notes in theoretical computer science, 229(5):75–95. Proceedings of the second workshop on mathematically structured functional programming (MSFP 2008)

[Jon83a] Jones CB (1983) Specification and design of (parallel) programs. In: IFIP Congress, pp 321–332

[Jon83b] Jones CB (1983) Tentative steps toward a development method for interfering programs. ACM Trans Program Lang Syst 5:596–619

[Jon90] Jones CB (1990) Systematic Software Development using VDM. Prentice Hall, Upper Saddle River

[Jon03a] Jones CB (2003) Operational semantics: concepts and their expression. Inf Process Lett 88(1–2):27–32

[Jon03b] Jones CB (2003) Wanted: a compositional approach to concurrency. Programming methodology. Springer, Berlin, pp 5–15

[Jon07] Jones CB (2007) Understanding programming language concepts via operational semantics. In: George C, Liu Z, Woodcock J (eds) Domain modeling and the duration calculus, international training school, advanced lectures, volume 4710 of lecture notes in computer science. Springer, Berlin, pp 177–235

[JR05] Jeffrey A, Rathke J (2005) Java Jr: fully abstract trace semantics for a core Java language. In: Sagiv S (ed) European symposium on programming, volume 3444 of lecture notes in computer science. Springer, Berlin, pp 423–438
[Kli11] Klin B (2011) Bialgebras for structural operational semantics: an introduction. Theor Comput Sci 412(38):5043–5069
[KLW95] Kifer M, Lausen G, Wu J (1995) Logical foundations of object-oriented and frame-based languages. J ACM 42(4):741–843
[KN06] Klein G, Nipkow T (2006) A machine-checked model for a Java-like language, virtual machine, and compiler. ACM Trans Program Lang Syst 28(4):619–695
[LF08] Leuschel M, Fontaine M (2008) Probing the depths of CSP-M: a new FDR-compliant validation tool. In: Liu S, Maibaum T, Araki K (eds) Formal methods and software engineering, volume 5256 of lecture notes in computer science. Springer, Berlin/Heidelberg, pp 278–297
[LLM07] Leavens GT, Rustan K, Leino M, Müller P (2007) Specification and verification challenges for sequential object-oriented programs. Formal Asp Comput 19(2):159–189
[LM05] Leino K, Müller P (2005) Modular verification of static class invariants. In: Fitzgerald J, Hayes I, Tarlecki A (eds) FM 2005: formal methods, volume 3582 of lecture notes in computer science. Springer, Berlin/Heidelberg, pp 597–597
[Loc10] Lochbihler A (2010) Verifying a compiler for Java threads. In: Gordon A (ed) Programming languages and systems, volume 6012 of lecture notes in computer science. Springer, Berlin/Heidelberg, pp 427–447
[Log09] Logozzo F (2009) Class invariants as abstract interpretation of trace semantics. Comput Lang Syst Struct 35(2):100–142
[Mes00] Meseguer J (2000) Rewriting logic and Maude: a wide-spectrum semantic framework for object-based distributed systems. In: Smith SF, Talcott CL (eds) Formal methods for open object-based distributed systems IV, IFIF TC6/WG6.1 (FMOODS 2000), volume 177 of IFIP conference proceedings. Kluwer Academic Publishers, Dordrecht, pp 89–119
[Mey00] Meyer B (2000) Object-oriented software construction, 2nd edn. Prentice Hall PTR, Upper Saddle River
[MF08] Matthews J, Findler RB (2008) An operational semantics for Scheme. J Funct Program 18(1):47–86
[Mil82] Milner R (1982) A calculus of communicating systems. Springer, New York
[Mil89] Milner R (1989) Communication and concurrency. Prentice Hall, Upper Saddle River
[MMT08] Maffeis S, Mitchell J, Taly A (2008) An operational semantics for JavaScript. In: Ramalingam G (ed) Programming languages and systems, volume 5356 of lecture notes in computer science, pp 307–325. Springer, Berlin/Heidelberg
[MN09] Mosses PD, New MJ (2009) Implicit propagation in structural operational semantics. Electr Notes Theor Comput Sci 229(4):49–66
[Mor94] Morgan C (1994) Programming from Specifications, 2nd edn. Prentice Hall, Upper Saddle River
[Mos02] Mosses PD (2002) Pragmatics of modular SOS. In Kirchner H, Ringeissen C (eds) Algebraic methodology and software technology, 9th international conference, AMAST 2002, Proceedings, volume 2422 of lecture notes in computer science. Springer, Berlin, pp 21–40
[Mos04a] Mosses PD (2004) Exploiting labels in structural operational semantics. Fundam Inform 60(1–4):17–31
[Mos04b] Mosses PD (2004) Modular structural operational semantics. J Log Algebr Program 60–61:195–228
[MPHL06] Müller P, Poetzsch-Heffter A, Leavens GT (2006) Modular invariants for layered object structures. Sci Comput Programm 62(3):253–286. Special issue on source code analysis and manipulation (SCAM 2005)
[MR11] Meseguer J, Rosu G (2011) The rewriting logic semantics project: a progress report. In: Owe O, Steffen M, Telle J (eds) Fundamentals of computation theory, volume 6914 of lecture notes in computer science. Springer, Berlin/Heidelberg, pp 1–37
[MTM97] Milner R, Tofte M, MacQueen D (1997) The definition of standard ML. MIT Press, Cambridge
[Nip03] Nipkow T (2003) Java bytecode verification. J Autom Reason 30(3–4):233–233
[NPW02] Nipkow T, Paulson LC, Wenzel M (2002) Isabelle/HOL—a proof assistant for higher-order logic, volume 2283 of LNCS. Springer, Berlin
[Owe08] Owens S (2008) A sound semantics for OCaml light. In: Drossopoulou S (ed) European symposium on programming (ESOP), volume 4960 of lecture notes in computer science, pp 1–15. Springer, Berlin
[PF07] Pop A, Fritzson P (2007) An Eclipse-based integrated environment for developing executable structural operational semantics specifications. In: Electronic notes in theoretical computer science, 175(1):71 – 75, 2007. Proceedings of the third workshop on structural operational semantics (SOS 2006)
[PHS06] Poetzsch-Heffter A, Schäfer J (2006) Modular specification of encapsulated object-oriented components. In: de Boer F, Bonsangue M, Graf S, de Roever W-P (eds) Formal methods for components and objects, volume 4111 of lecture notes in computer science, pp 313–341. Springer, Berlin/Heidelberg
[PHS07] Poetzsch-Heffter A, Schäfer J (2007) A representation-independent behavioral semantics for object-oriented components. In: Bonsangue M, Johnsen E (eds) Formal methods for open object-based distributed systems, volume 4468 of lecture notes in computer science. Springer, Berlin/Heidelberg, pp 157–173
[Plo81] Plotkin GD (1981) A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University
[Plo04] Plotkin GD (2004) A structural approach to operational semantics. J Log Algebr Program 60–61:17–139
[Rei95] Reichel H (1995) An approach to object semantics based on terminal co-algebras. Math Struct Comput Sci 5(02):129–152
[Rey98] Reynolds JC (1998) Theories of programming languages. Cambridge University Press, Cambridge
[Smi00] Smith G (2000) The Object-Z specification language. Springer, Berlin
[Spi92] Spivey JM (1992) The Z notation: a reference manual, 2nd edn. Prentice Hall, Upper Saddle River
[SSB01] Stark RF, Schmid J, Börger E (2001) Java and the Java Virtual Machine: definition, verification, validation. Springer, Berlin
[SSL08] Silva L, Sampaio A, Liu Z (2008) Laws of object-orientation with reference semantics. In: IEEE international conference on software engineering and formal methods, Los Alamitos, CA, USA, IEEE Computer Society, pp 217–226
[Stä05] Stärk RF (2005) Formal specification and verification of the C# thread model. Theor Comput Sci 343(3):482–508
[Str97] Stroustrup B (1997) The C++ Programming Language. Addison-Wesley, Reading
[TP97] Turi D, Plotkin GD (1997) Towards a mathematical operational semantics. In: IEEE symposium on logic in computer science (LICS), pp 280–291. IEEE Computer Society

[vG01] van Glabbeek R (2001) The linear time: branching time spectrum I. In: Bergstra J, Ponse A, Smolka S (eds) Handook of process algebra. North-Holland, Amsterdam, pp 3–99
[VMO04] Verdejo A, Mart-Oliet N (2004) Implementing CCS in Maude 2. Electronic Notes Theor Comput Sci 71:282–300
[VMO06] Verdejo A, Mart-Oliet N (2006) Executable structural operational semantics in Maude. J Logic Algebraic Programm 67(1–2):226–293
[WNST06] Wasserrab D, Nipkow T, Snelting G, Tip F (2006) An operational semantics and type safety proof for multiple inheritance in C++. In: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, (OOPSLA 06), pp 345–362. ACM

Received 12 October 2011
Revised 4 April 2012
Accepted 27 June 2012 by Dong Jin Song
Published online 21 August 2012